CAPÍTULO 2

Solución de problemas mediante C++

TEMAS

2.1 INTRODUCCIÓN A C++

Un programa bien diseñado se construye utilizando una filosofía similar a la usada para edificar un edificio bien diseñado: no sólo sucede; depende de una planeación y ejecución cuidadosa si el diseño final ha de cumplir su propósito pretendido. Del mismo modo en que una parte integral del diseño de un edificio es su estructura, lo mismo sucede con un programa.

Los programas cuyas estructuras consisten de segmentos interrelacionados, organizados en un orden lógico y fácilmente comprensible para formar una unidad integrada y completa, se conocen como **programas modulares** (figura 2.1). Los programas modulares son notoriamente más fáciles de desarrollar, corregir y modificar que los programas construidos de alguna otra manera. En terminología de programación, los segmentos más pequeños usados para construir un programa modular se conocen como **módulos**.

Cada módulo está diseñado y desarrollado para realizar una tarea específica y en realidad es un subprograma pequeño en sí mismo. Un programa C++ completo se construye combinando tantos módulos como sea necesario para producir el resultado deseado. La ventaja de la construcción modular es que el diseño general del programa puede desarrollarse antes de escribir cualquier módulo individual. Una vez que se finalizan los requerimientos para cada módulo, los módulos pueden programarse e integrarse dentro del programa general conforme se completan.

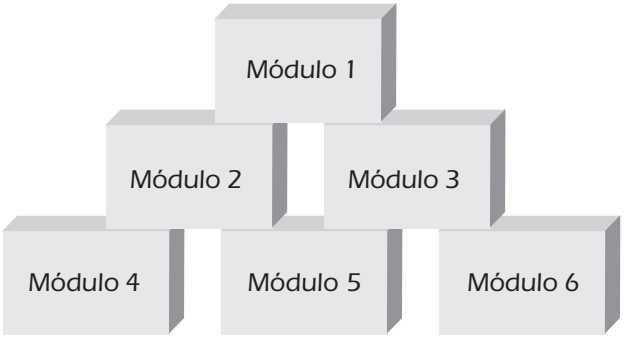


Figura 2.1 *Un programa bien diseñado se construye usando módulos.*

En C++, los módulos pueden ser clases o funciones. Ayuda pensar en una **función** como una máquina pequeña que transforma los datos que recibe en un producto terminado. Por ejemplo, la figura 2.2 ilustra una función que acepta dos números como entradas y multiplica los dos números para producir una salida. Como se muestra, la interfaz para la función son sus entradas y resultados. El proceso de convertir las entradas en resultados se encapsula y oculta dentro de la función. A este respecto, la función puede considerarse como una sola unidad que proporciona una operación de propósito especial. Una analogía similar es apropiada para una **clase**.

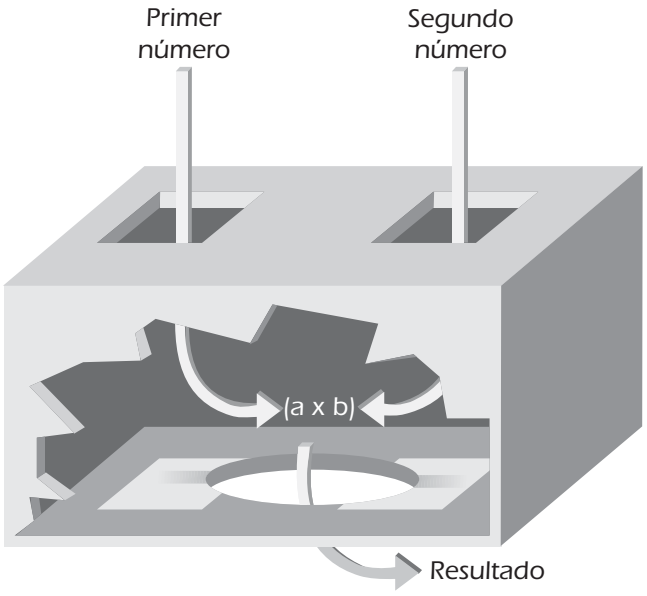


Figura 2.2 *Una función multiplicadora.*

Una **clase** es una unidad más complicada que una función, debido a que contiene tanto datos como funciones apropiadas para manipular los datos. Por tanto, a diferencia de una función, la cual se usa para encapsular un conjunto de operaciones, una clase encapsula tanto datos como uno o más conjuntos de operaciones. Por ello, cada clase contiene todos los elementos requeridos para la entrada, salida y procesamiento de sus objetos y puede considerarse como una pequeña fábrica que contiene materia prima (los datos) y máquinas (las funciones). Al principio, nos interesaremos de manera predominante en el módulo de función más básico.

Un requerimiento importante para diseñar una buena función o clase es darle un nombre que le transmita al lector alguna idea de lo que hace la función o clase. Los nombres admitidos para funciones y clases también se usan para nombrar otros elementos del lenguaje C++ y se denominan de manera colectiva ***identificadores***. Los identificadores pueden formarse por cualquier combinación de letras, dígitos o subrayados (_) seleccionados de acuerdo con las siguientes reglas:

1. El primer carácter del nombre debe ser una letra o subrayado (_).
2. Sólo pueden seguir a la letra inicial, letras, dígitos o subrayados. No se permiten espacios en blanco; se usa el subrayado para separar palabras en un nombre consistente de múltiples palabras o se usan mayúsculas para la primera letra de una o más palabras.
3. El nombre de una función no puede ser una de las palabras clave enumeradas en la tabla 2.1. (Una **palabra clave** es una palabra que es apartada por el lenguaje para un propósito especial y sólo puede usarse en una manera especificada.)[1]
4. El número máximo de caracteres en una función es 1024.[2]

[1]Las palabras clave en C también son palabras reservadas, lo cual significa que deben usarse sólo para el propósito especificado. Intentar usarlas para cualquier otro propósito generará un mensaje de error.

[2]Éste es el mínimo requerido por el estándar ANSI.

Tabla 2.1 Palabras clave

auto	delete	goto	public	this
break	do	if	register	template
case	double	inline	return	typedef
catch	else	int	short	union
char	enum	long	signed	unsigned
class	extern	new	sizeof	virtual
const	float	overload	static	void
continue	for	private	struct	volatile
default	friend	protected	switch	while

Son ejemplos de identificadores válidos de C++:

```
GradARad     intersección    sumarNum    pendiente
bess         mult_dos        HallarMax   densidad
```

Son ejemplos de identificadores inválidos:

1AB3 (empieza con un número, lo cual viola la regla 1)

E*6 (contiene un carácter especial, lo cual viola la regla 2)

while (ésta es una palabra clave, lo cual viola la regla 3)

Además de apegarse a las reglas de identificadores de C++, el nombre de la función siempre debe ser seguido por paréntesis. (La razón para esto se explicará más adelante.) Además, un buen nombre de función deberá ser ***mnemónico***, esto es, una palabra o nombre designado como un auxiliar para la memoria. Por ejemplo, el nombre de función `GradARad()` (note que hemos incluido los paréntesis requeridos después del identificador, lo cual lo señala con claridad como un nombre de función) es un mnemónico si es el nombre de una función que convierte grados a radianes. Aquí, el nombre en sí ayuda a identificar lo que hace la función.

Son ejemplos de nombres de función válidos que no son mnemónicos:

```
fácil()    c3po()    r2d2()    la fuerza()    miguel()
```

Los nombres de función que no son mnemónicos no deberán usarse porque no transmiten información sobre lo que hace la función.

Además, los nombres de función se pueden escribir como una mezcla de letras mayúsculas y minúsculas. Esto se está volviendo cada vez más común en C++, aunque no es absolutamente necesario. Los identificadores escritos por completo en mayúsculas por lo general se reservan para constantes simbólicas, un tema que se cubre en la sección 3.5.

Por añadidura, C++ es un lenguaje ***sensible al uso de mayúsculas y minúsculas***. Esto significa que el compilador distingue entre letras mayúsculas y minúsculas. Por tanto, en C++, los nombres TOTAL, total y TotaL, representan tres nombres distintos.

La función `main()`

Una ventaja distintiva de usar funciones y clases en C++ es que la estructura total del programa en general, y de módulos individuales en particular, puede planearse con anticipación, incluyendo provisiones para probar y verificar la operación de cada módulo. Cada función y clase puede escribirse entonces para cumplir con su objetivo pretendido.

Para facilitar la colocación y ejecución ordenada de los módulos, cada programa C++ debe tener una y sólo una función llamada `main()`. La función `main()` se conoce como una ***función controladora***, porque controla, o indica, a los otros módulos la secuencia en la que tienen que ejecutarse (figura 2.3).[3]

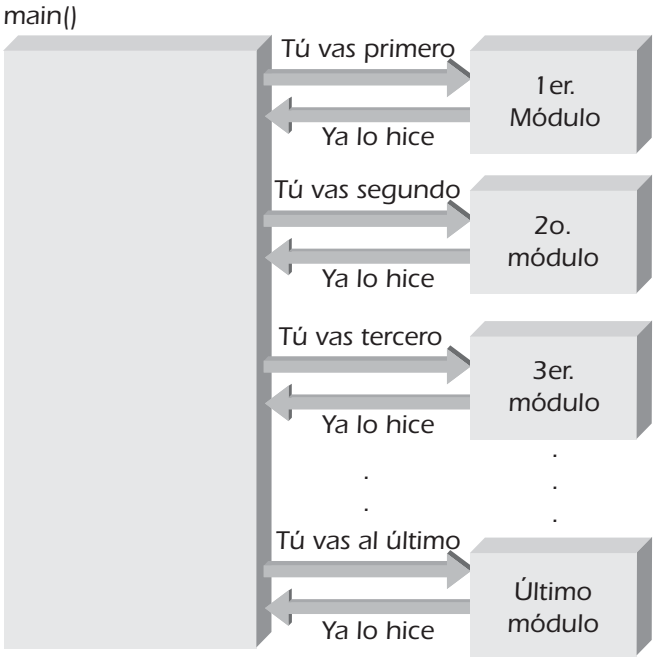


Figura 2.3 *La función* `main()` *dirige a todas las otras funciones.*

La figura 2.4 ilustra una estructura para la función `main()`. La primera línea de la función, en este caso `int main()`, se conoce como ***línea de encabezado de la función***. Una línea de encabezado de la función, la cual siempre es la primera línea de una función, contiene tres fragmentos de información:[4]

1. Qué tipo de dato, si hay alguno, es devuelto por la función.
2. El nombre de la función.
3. Qué tipo de dato, si hay alguno, es enviado a la función.

La palabra clave antes del nombre de la función define el tipo de valor que devuelve la función cuando ha completado su operación. Cuando se coloca antes del nombre de la función,

[3]Los módulos ejecutados a partir de `main()` pueden, a su vez, ejecutar otros módulos. Cada módulo, sin embargo, siempre regresa al módulo que inició su ejecución. Esto se aplica incluso a `main()`, la cual regresa el control al sistema operativo cuando fue iniciado `main()`.

[4]Un método de clase también debe empezar con una línea de encabezado que se apega a estas mismas reglas.

la palabra clave `int` (véase la tabla 2.1) indica que la función devolverá el valor de un número entero. Del mismo modo, cuando los paréntesis que siguen al nombre de la función están vacíos, significa que no se transmitirán datos a la función cuando se esté ejecutando. (Los datos transmitidos a una función al momento de ejecutarla se llaman ***argumentos*** de la función.) Las llaves { y }, determinan el principio y el fin del cuerpo de la función y encierran las instrucciones que la componen. Las instrucciones dentro de las llaves determinan lo que hace la función. Cada instrucción dentro de la función debe terminar con un punto y coma (;).

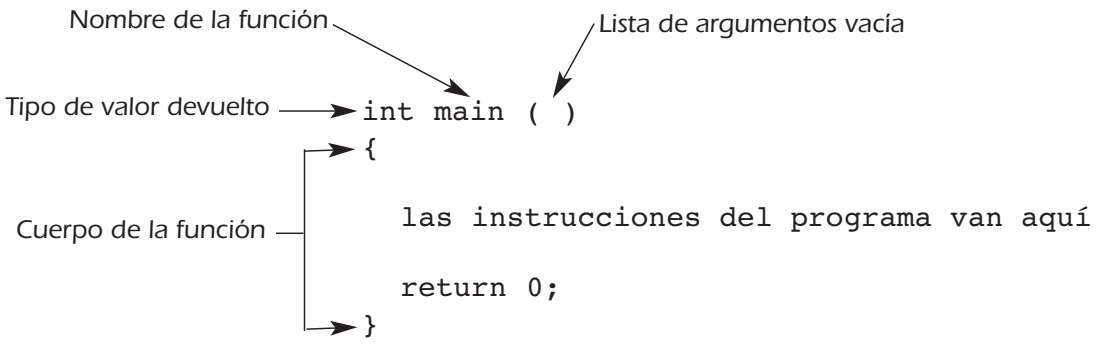


Figura 2.4 *La estructura de una función* `main()`.

Se pueden nombrar y escribir muchas de las funciones C++ personalizadas. De hecho, el resto de este libro se ocupa de manera principal de las instrucciones requeridas para elaborar funciones útiles y la forma de combinar funciones y datos en clases y programas útiles. Cada programa, sin embargo, debe tener una y sólo una función `main()`. Hasta que aprenda cómo pasar y regresar los datos de una función (los temas del capítulo 6), la línea de encabezado ilustrada en la figura 2.4 servirá para todos los programas que sea necesario escribir. Para programas simples, hay que considerar que las primeras dos líneas (que se explican con más detalle en el capítulo 6):

```
int main()
{
```

tan sólo señalan que "el programa comienza aquí", mientras las últimas dos líneas

```
  return 0;
}
```

señalan el final del programa. Por suerte, muchas funciones y clases útiles ya se presentan escritas. Ahora se verá cómo usar un objeto creado a partir de una de estas clases para crear nuestro primer programa funcional en C++.

El objeto `cout`

Uno de los objetos más versátiles y más usados en C++ se llama `cout` (pronunciado "si out"). Este objeto, cuyo nombre se deriva de Console OUTput, es un objeto de salida que envía datos introducidos en él al dispositivo estándar de salida.[5] Para la mayoría de los sis-

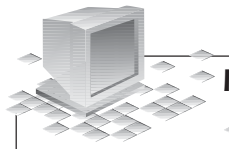
[5] El objeto `cout` se crea de manera formal a partir de la clase `ostream`, la cual se describe con más detalle en el capítulo 7.

temas, este dispositivo de salida es una pantalla de video. El objeto `cout` muestra en el monitor cualquier cosa enviada a él. Por ejemplo, si se envían los datos `¡Hola mundo!` a `cout`, estos datos se imprimen (o despliegan) en la pantalla de la terminal. Los datos `¡Hola mundo!` se envían al objeto `cout` encerrando el texto entre comillas, `"poner cadena aquí"`, y colocando el símbolo de inserción `<<` ("enviar a"), antes del mensaje y después del nombre del objeto, como se muestra en la figura 2.5.

```
cout << "¡Hola mundo!";
```

Figura 2.5 *Envío de un mensaje a* `cout`.

Ahora juntemos todo esto en un programa C++ funcional que pueda ejecutarse en su computadora. Considere el programa 2.1.



Programa 2.1

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

int main()
{
  cout << "Hola mundo!";

  return 0;
}
```

La primera línea del programa,

```
#include <iostream>
```

es un comando preprocesador que utiliza la palabra reservada `include`. Los comandos preprocesadores comienzan con un signo de número (#) y ejecutan alguna acción antes que el compilador traduzca el programa fuente a código de máquina. De manera específica, el comando preprocesador `#include` causa que el contenido del archivo invocado, en este caso el archivo `iostream`, sea insertado en donde el comando `#include` aparezca en el programa. El `iostream` es una parte de la biblioteca estándar que contiene, entre otro código, dos clases nombradas `istream` y `ostream`. Estas dos clases proporcionan las declaraciones de datos y métodos utilizados para la entrada y salida de datos, respectivamente. El archivo `iostream` se conoce como un **archivo de encabezado** debido a que siempre se coloca una referencia a él en la parte superior, o cabeza, de un programa C++ usando el comando #include. Puede ser que se pregunte qué tiene que ver el archivo `iostream` con este programa simple. La respuesta es que el objeto `cout` se crea a partir de la clase `ostream`. Por tanto, el archivo de encabezado `iostream` debe incluirse en todos los programas que usen `cout`. Como se indicó en el programa 2.1, los comandos preprocesadores no terminan con un punto y coma.

Punto de Información

¿Qué es la sintaxis?

La **sintaxis** de un lenguaje de programación es el conjunto de reglas para formular instrucciones gramaticalmente correctas en un lenguaje. En la práctica esto significa que una instrucción en C++ con sintaxis correcta tiene la forma apropiada especificada para el compilador. Como tal, el compilador aceptará la instrucción y no generará un mensaje de error.

Debe señalarse que una instrucción o programa individual puede ser correcto desde el punto de vista sintáctico, pero incorrecto desde el punto de vista lógico. Una instrucción o programa así estaría estructurado de manera correcta pero produciría un resultado incorrecto. Esto es similar a un enunciado en español que es correcto desde el punto de vista gramatical pero no tiene sentido. Por ejemplo, aunque el enunciado "El árbol es un gato andrajoso" es gramaticalmente correcto, no tiene sentido.

Después del comando preprocesador `include` hay una instrucción que contiene la palabra reservada `using`. La instrucción,

```
using namespace std;
```

le dice al compilador dónde buscar para encontrar los archivos de encabezado en ausencia de cualquier designación explícita adicional. Se puede pensar en un espacio de nombres como un archivo de código fuente al que tiene acceso el compilador cuando busca clases o funciones preescritas. Debido a que el archivo de encabezado `iostream` está contenido dentro de un espacio de nombres llamado `std`, el compilador usará de manera automática el objeto `cout` de `iostream` a partir de este espacio de nombres siempre que se haga referencia a `cout`. El uso de espacio de nombres le permite crear sus propias clases y funciones con los mismos nombres provistos por la biblioteca estándar y colocarlos en espacios de nombres invocados de manera diferente. Puede señalar al programa cuál clase o función usar al indicar el espacio de nombres donde desea que el compilador busque la clase o función.

La instrucción `using` es seguida por el inicio de la función `main()` del programa. Esta función comienza con la línea de encabezado desarrollada al principio de esta sección. El cuerpo de la función, entre llaves, consta de sólo dos instrucciones. La primera instrucción en `main()` pasa un mensaje al objeto `cout`. El mensaje es la cadena `"¡Hola mundo!"`.

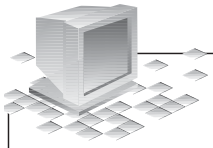
Debido a que `cout` es un objeto de una clase preescrita, no tenemos que escribirlo; está disponible para su uso con sólo activarlo en forma correcta. Como todos los objetos de C++, `cout` sólo puede ejecutar ciertas acciones bien definidas. Para `cout`, la acción es ensamblar datos para mostrar la salida. Cuando se pasa una cadena de caracteres a `cout`, el objeto asegura que la cadena se despliegue de manera correcta en su monitor, como se muestra en la figura 2.6.

```
¡Hola mundo!
```

Figura 2.6 *La salida del programa 2.1.*

Las cadenas en C++ son cualquier combinación de letras, números y caracteres especiales encerrados entre comillas ("colocar cadena aquí"). Las comillas se usan para delimitar (marcar) el comienzo y el final de la cadena y no se consideran parte de la cadena. Por tanto, la cadena de caracteres que forma el mensaje enviado a `cout` debe encerrarse entre comillas, como se ha hecho en el programa 2.1.

Escribiremos otro programa para ilustrar la versatilidad de `cout`. Lea el programa 2.2 para determinar qué hace.



Programa 2.2

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

int main()
{
  cout << "Computadoras, computadoras por todos lados";
  cout << "\n   tan lejos como pueda llegar C";

  return 0;
}
```

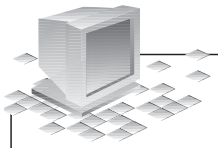
Cuando se ejecuta el programa 2.2 se despliega lo siguiente:

```
Computadoras, computadoras por todos lados
   tan lejos como pueda llegar C
```

Tal vez se pregunte por qué no apareció `\n` en la salida. Los dos caracteres \ y n, cuando se usan juntos, se llaman secuencia de escape para una línea nueva. Le indican a `cout` que envíe instrucciones al dispositivo de salida para iniciar una línea nueva. En C++, el carácter de diagonal inversa (\) proporciona un "escape" de la interpretación normal del carácter que la sigue, alterando el significado del siguiente carácter. Si se omitiera la diagonal inversa de la segunda instrucción `cout` en el programa 2.2, la n se imprimiría como la letra n y el programa desplegaría:

```
Computadoras, computadoras por todos lados   tan lejos como pue-
da llegar C
```

Pueden colocarse secuencias de escape para una línea nueva en cualquier parte dentro del mensaje que se transmite a `cout`. Vea si puede determinar el despliegue producido por el programa 2.3.



Programa 2.3

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

int main()
{
  cout << "Computadoras por todos lados\n tan lejos como\n\npueda llegar C";

  return 0;
}
```

La salida para el programa 2.3 es:

```
Computadoras por todos lados tan lejos como
 puede llegar C

puedo ver
```

Ejercicios 2.1

1. Indique si los siguientes son nombres de función válidos. Si son válidos, indique si son nombres mnemónicos. (Recuerde que un nombre de función mnemónico transmite alguna idea sobre el propósito de la función.) Si son nombres inválidos, indique por qué.

```
poder     densidad   m1234$          ampnuevo   1234    abcd
total     tangente   valAbs          computado  b34a    34ab
voltios$  a2B3       while           valMin     seno    $seno
coseno    velocidad  distancianeta   suma       return  pila
```

2. Suponga que se han escrito las siguientes funciones:

   `obtenerLargo(), obtenerAncho(), calcArea(), mostrarArea()`

   a. A partir de los nombres de las funciones, ¿qué piensa que podría hacer cada función?
   b. ¿En qué orden piensa que una función `main()` podría ejecutar estas funciones (con base en sus nombres)?

3. Suponga que se han escrito las siguientes funciones:

   `velocidad(), distancia(), aceleración()`

   A partir de los nombres de las funciones, ¿qué piensa que podría hacer cada función?

4. Determine nombres para funciones que hagan lo siguiente:
   a. Encontrar el promedio de un conjunto de números.
   b. Encontrar el área de un rectángulo.

c. Encontrar el valor mínimo en un conjunto de números.
d. Encontrar la densidad de una puerta de acero.
e. Clasificar un conjunto de números de menor a mayor.

5. Del mismo modo en que la palabra clave `int` se usa para indicar que una función devolverá un número entero, las palabras clave `void`, `char`, `float` y `double` se usan para indicar que una función no devolverá ningún valor o devolverá un carácter, un número en punto flotante en precisión simple, y un número en precisión doble, respectivamente. Usando esta información, escriba líneas de encabezado para una función `main()` que no recibirá argumentos pero que devolverá:

a. ningún valor
b. un carácter
c. un número de punto flotante en precisión simple
d. un número en precisión doble

6. a. Usando `cout`, escriba un programa en C++ que despliegue su nombre en una línea, su domicilio en una segunda línea, y su ciudad, estado y código postal en una tercera línea.

b. Ejecute el programa que ha escrito para el ejercicio 6a en una computadora. (*NOTA:* Debe entender los procedimientos para introducir y ejecutar un programa C++ en la computadora particular que esté usando.)

7. a. Escriba un programa en C++ para desplegar lo siguiente:

```
La cosecante de un ángulo
   es igual a uno dividido entre
     el seno del ángulo.
```

b. Compile y ejecute el programa que ha escrito para el ejercicio 7a en una computadora.

8. a. ¿Cuántas instrucciones `cout` usaría para desplegar lo siguiente:

```
Grados          Radianes
 0              0.0000
90              1.5708
180             3.1416
270             4.7124
360             6.2832
```

b. ¿Cuál es el número mínimo de instrucciones cout que podrían usarse para imprimir la tabla en el ejercicio 8a?
c. Escriba un programa en C++ completo para producir la salida ilustrada en el ejercicio 8a.
d. Ejecute el programa que ha escrito para el ejercicio 8c en una computadora.

9. En respuesta a una secuencia de escape de línea nueva, `cout` coloca el siguiente carácter desplegado al principio de una línea nueva. Esta colocación del siguiente carácter en realidad representa dos operaciones distintas. ¿Cuáles son?

10. a. Suponiendo un compilador que no es sensible al uso de mayúsculas y minúsculas, determine cuáles de estos nombres de unidades de programa son equivalentes:

```
PROMEDIO   promedio   MODO      BESSEL     Modo
Total      besseL     TeMp      Densidad   TEMP
densiDAD   MEDIA      total     media      moDO
```

b. Vuelva a hacer el ejercicio 10a suponiendo que el compilador es sensible al uso de mayúsculas y minúsculas.

Ejercicios para estructurar un proyecto

La mayor parte de los proyectos, tanto de programación como de otro tipo, por lo general pueden estructurarse en subtareas o unidades de actividad más pequeñas. Estas subtareas más pequeñas a menudo pueden delegarse a diferentes personas, de modo que cuando todas las tareas se terminan e integran, el proyecto o programa está completo. Para los ejercicios 11 a 16, determine un conjunto de subtareas que, unidas, completen el proyecto. Tome en cuenta que hay muchas soluciones posibles para cada ejercicio. El único requisito es que el conjunto de subtareas seleccionadas, cuando se junten, completen la tarea requerida.

NOTA: El propósito de estos ejercicios es que considere las diferentes formas en que pueden estructurarse tareas complejas. Aunque no hay una solución correcta para estos ejercicios, hay soluciones incorrectas y soluciones que son mejores que otras. Una solución incorrecta es aquella que no especifica por completo la tarea. Una solución es mejor que otra si identifica con más claridad o más facilidad lo que debe hacerse.

11. Se le dio la tarea de cablear e instalar luces en el ático de su casa. Determine un conjunto de subtareas que, unidas, harán que logre esto. (*Sugerencia:* La primera subtarea debería determinar la colocación de las lámparas.)

12. Se le da el trabajo de preparar una comida completa para cinco personas el próximo fin de semana. Determine un conjunto de subtareas que, unidas, harán que puedan lograr esto. (*Sugerencia:* Una subtarea, no necesariamente la primera, debería ser comprar los alimentos.)

13. Usted es un estudiante de segundo año de bachillerato y está planeando ir a una escuela superior para estudiar ingeniería eléctrica después de graduarse. Enumere un conjunto de objetivos principales que debe cumplir para alcanzar esta meta. (*Sugerencia:* Una subtarea es "Determinar los cursos adecuados que deberá tomar".)

14. Se le ha dado el trabajo de cultivar un huerto. Determine un conjunto de subtareas para lograr esto. (*Sugerencia:* Una de las subtareas sería planear la disposición del huerto.)

15. Usted es responsable de planificar y organizar un viaje de la familia a un campamento este verano. Enumere un conjunto de subtareas que, unidas, lo lleven a alcanzar este objetivo con éxito. (*Sugerencia:* Una subtarea sería seleccionar la ubicación del campamento.)

16. a. Un laboratorio nacional de exámenes médicos desea un sistema de cómputo nuevo para analizar los resultados de sus exámenes. El sistema debe ser capaz de procesar los resultados diarios. Además, el laboratorio requiere recuperar y obtener un informe impreso de todos los resultados que cumplen con ciertos cri-

terios, por ejemplo, todos los resultados obtenidos por un medico particular o todos los resultados obtenidos para los hospitales en un estado particular. Determine tres o cuatro unidades del programa en los que podría tratarse por separado este sistema. (*Sugerencia:* Una unidad del programa posible es "Preparar los resultados diarios" para crear los informes de cada día.)

b. Suponga que alguien introduce datos incorrectos para un resultado de examen particular, lo cual se descubre después que los datos se han introducido y almacenado en el sistema. ¿Qué unidad del programa es necesaria para corregir este problema? Discuta por qué una unidad del programa como ésta podría requerirse o no en la mayor parte de los sistemas.

c. Suponga que existe una unidad del programa que permite a un usuario alterar o cambiar datos que se han introducido y almacenado en forma incorrecta. Discuta la necesidad de incluir un "rastreo de auditoría" que permitiría una reconstrucción posterior de los cambios hechos, cuándo se hicieron y quién los hizo.

2.2 ESTILO DE PROGRAMACIÓN

Los programas en C++ empiezan la ejecución al principio de la función `main()`. Debido a que un programa sólo puede tener un punto de inicio, todo programa en lenguaje C++ debe contener una y sólo una función `main()`. Como hemos visto, todas las instrucciones que forman la función `main()` se incluyen luego dentro de las llaves que siguen al nombre de la función. Aunque la función `main()` debe estar presente en todos los programas C++, éste no requiere que la palabra main, los paréntesis `()` o las llaves `{}` se coloquen en alguna forma particular. La forma usada en la sección anterior,

```
int main()
{
  las instrucciones del programa van aquí;

  return 0;
}
```

se eligió sólo por claridad y facilidad en la lectura del programa. Si una de las instrucciones del programa usa el objeto `cout`, debe incluirse el archivo de encabezado `iostream`, al igual que la instrucción `using namespace std;`. Por ejemplo, la siguiente forma general de una función `main()` también funcionaría:

```
int main
 (
 ) { primera instrucción; segunda instrucción;
          tercera instrucción; cuarta
instrucción;
return 0;}
```

Note que puede ponerse más de una instrucción en una línea, o una instrucción puede escribirse en más de una línea. Con excepción de las cadenas, comillas, identificadores y palabras clave, C++ ignora todo el espacio en blanco. (El espacio en blanco se refiere a cualquier combinación de uno o más espacios en blanco, tabuladores o líneas nuevas.) Por

ejemplo, cambiar el espacio en blanco en el programa 2.1 y asegurarse que no se divide la cadena ¡Hola mundo! en dos líneas produce el siguiente programa válido:

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

int main
(
){
cout <<
"¡Hola mundo!";
 return 0;
}
```

Aunque esta versión de `main()` funciona, es un ejemplo de un estilo de programación pobre en extremo. Es difícil de leer y entender. Para legibilidad, la función `main()` siempre deberá escribirse en forma estándar como:

```
int main()
{
  las instrucciones del programa van aquí;

  return 0;
}
```

En esta forma estándar el nombre de la función empieza en la columna 1 y se coloca con los paréntesis requeridos en una sola línea. La llave de apertura del cuerpo de la función sigue en la siguiente línea y se coloca bajo la primera letra de la línea que contiene el nombre de la función. Del mismo modo, la llave que cierra la función se coloca sola en la columna 1 como la última línea de la función. Esta estructura sirve para resaltar la función como una unidad independiente.

Dentro de la función en sí, todas las instrucciones del programa tienen una sangría de por lo menos dos espacios. La sangría es otra buena práctica de programación, en especial si se usa la misma sangría para grupos similares de instrucciones. Revise el programa 2.2 para ver que se usó la misma sangría para ambas llamadas al objeto `cout`.

Mientras progresa en su comprensión y dominio de C++, desarrollará sus propios estándares para las sangrías. Sólo tenga en cuenta que la forma final de sus programas deberán ser consistentes y servir siempre como un auxiliar para la lectura y comprensión de sus programas.

Comentarios

Los comentarios son observaciones explicativas que se hacen dentro de un programa. Cuando se usan con cuidado, los comentarios pueden ser muy útiles para aclarar de qué trata el programa completo, qué pretende lograr un grupo específico de instrucciones o qué pretende hacer una línea. C++ acepta dos tipos de comentarios: de línea y de bloque. Ambos tipos de comentarios pueden colocarse en cualquier parte dentro de un programa y no tienen efecto

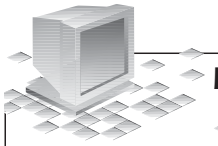
en la ejecución del mismo. El compilador ignora todos los comentarios; están ahí estrictamente para la conveniencia de cualquiera que lea el programa.

Un ***comentario de línea*** empieza con dos diagonales (`//`) y continúa hasta el final de la línea. Por ejemplo, las siguientes líneas son comentarios de línea:

```
// éste es un comentario
// este programa imprime un mensaje
// este programa calcula una raíz cuadrada
```

Los símbolos `//`, sin espacio en blanco entre ellos, designan el inicio del comentario de línea. El final de la línea en la que está escrito el comentario designa el final del comentario.

Un comentario de línea puede escribirse ya sea en una sola línea o al final de la misma línea que contiene una instrucción del programa. El programa 2.4 ilustra el uso de comentarios de línea dentro de un programa.



Programa 2.4

```
// este programa despliega un mensaje
#include <iostream>
using namespace std;

int main()
{
  cout << "¡Hola mundo!"; // esto produce el despliegue

  return 0;
}
```

El primer comentario aparece en una línea aparte al principio del programa y describe lo que hace el programa. Por lo general ésta es una buena ubicación para incluir un comentario breve que describa el propósito del programa. Si se requieren más comentarios, puede colocarse uno por línea. Por tanto, cuando un comentario es demasiado largo para que quepa en una línea, puede separarse en dos o más comentarios de línea, con cada comentario separado precedido por el símbolo de doble diagonal establecido `//`. El comentario

```
// este comentario es inválido porque
   se extiende en dos líneas
```

producirá un mensaje de error de C++ en su computadora. Este comentario es correcto cuando se escribe así

```
// este comentario se usa para ilustrar un
// comentario que se extiende en dos líneas
```

Sin embargo, los comentarios que abarcan dos o más líneas se escriben de manera más conveniente como comentarios en bloque tipo C que como múltiples comentarios de línea. Los **comentarios en bloque** comienzan con los símbolos `/*` y terminan con los símbolos `*/`. Por ejemplo,

```
/* Éste es un comentario en bloque que
    abarca
    tres líneas */
```

En C++, la estructura del programa pretende que sea legible y comprensible, haciendo innecesario el uso de comentarios extensos. Esto se refuerza si los nombres de función, clase y variable, que se describen en el siguiente capítulo, son seleccionados con cuidado para transmitir su significado a cualquiera que lea el programa. Sin embargo, si el propósito de una función, clase o instrucción aún no está claro a partir de su estructura, nombre o contexto, incluya comentarios donde se necesiten aclaraciones. El código oscuro, sin comentarios, es una señal segura de mala programación y se hace presente cuando el programa debe ser sometido a mantenimiento o ser leído por otros. Del mismo modo, los comentarios excesivos también son una señal de mala programación, porque implican que hubo un razonamiento insuficiente para hacer que el código se explicara por sí solo. Por lo general, cualquier programa que escriba deberá comenzar con una serie de comentarios iniciales al programa que incluyan una descripción breve de éste, su nombre y la fecha de la última modificación. Por consideraciones de espacio, y debido a que todos los programas en este texto fueron escritos por el autor, sólo se usarán comentarios iniciales para descripciones breves de los programas cuando no se proporcionen como parte del texto descriptivo acompañante.

Ejercicios 2.2

1. **a.** ¿Funcionará el siguiente programa?

```
#include <iostream>
using namespace std;
int main() {cout << "¡Hola mundo!"; return 0;}
```

   b. ¿Por qué el programa dado en el ejercicio 1a no es un buen programa?

2. Vuelva a escribir los siguientes programas para que se ajusten a una buena práctica de programación y una sintaxis correcta.

   a.
```
#include <iostream>
  int main(
  ){
  cout          <<
  "El momento ha llegado"
  ; return 0;}
```

   b.
```
#include <iostream>
using namespace std;
  int main
  (    ){cout << "Newark es una ciudad\n";cout <<
  "En Nueva Jersey\n"; cout <<
  "También es una ciudad\n"
  ; cout << "En Delaware\n"
  ; return 0;}
```

c.
```
#include <iostream>
using namespace std;
 int main() {cout << Leer un programa\n";cout <<
 "es mucho más fácil\n"
 ; cout << "si se usa una forma estándar para main\n")
 ; cout
 <<"y cada instrucción se escribe\n";cout
 <<           "en una línea separada\n")
 ; return 0;}
```

d.
```
#include <iostream.h>
using namespace std;
 int main
 (    ){ cout << "Todo programa en C++"
 ; cout
 <<"\ndebe tener una y sólo una"
 ;
 cout << "función main"
 ;
 cout <<
 "\n la secuencia de escape de caracteres")
 ; cout <<
   "\npara una línea nueva puede colocarse en cualquier
      parte"
 ; cout
 <<"\n dentro del mensaje transmitido a cout"
 ; return 0;}
```

3. a. Cuando se usa en un mensaje, el carácter de diagonal inversa altera el significado del carácter que le sigue inmediatamente. Si deseamos imprimir el carácter de diagonal inversa, tendríamos que proporcionan a `cout` una secuencia de escape para que interprete normalmente la diagonal inversa. ¿Qué carácter piensa que se usa para alterar la forma en que se interpreta un solo carácter de diagonal inversa?

b. Usando su respuesta al ejercicio 3a, escriba la secuencia de escape para imprimir una diagonal inversa.

4. a. Una ***señal (token*** en inglés**)** de un lenguaje de computadora es cualquier secuencia de caracteres que, como una unidad, sin caracteres que intervengan o espacios en blanco, tiene un significado único. Usando esta definición de token, determine si las secuencias de escape, nombres de función y las palabras clave enumeradas en la tabla 1.2 son tokens del lenguaje C++.

b. Analice si agregar un espacio en blanco a un mensaje lo altera. Analice si los mensajes pueden considerarse tokens de C++.

c. Utilizando la definición de token que se dio en el ejercicio 4a, determine si la siguiente instrucción es verdadera: "Excepto por las tokens del lenguaje, C++ ignora todos los espacios en blanco."

2.3 TIPOS DE DATOS

El objetivo de todos los programas es procesar datos, sean numéricos, alfabéticos, de audio o de video. Es fundamental para este objetivo la clasificación de los datos en tipos específicos. Por ejemplo, calcular la trayectoria de un cohete requiere operaciones matemáticas con datos numéricos, y alfabetizar una lista de nombres requiere operaciones de comparación con datos basados en caracteres. Además, algunas operaciones no son aplicables a ciertos tipos de datos. Por ejemplo, no tiene sentido sumar nombres. Para impedir que los programadores intenten ejecutar una operación inapropiada, C++ sólo permite que se ejecuten ciertas operaciones con ciertos tipos de datos.

Los tipos de dato permitidos y las operaciones apropiadas definidas para cada tipo se conocen como tipo de dato. De manera formal, un **tipo de dato** se define como un conjunto de valores y un conjunto de operaciones que pueden aplicarse a estos valores. Por ejemplo, el conjunto de todos los números enteros constituye un conjunto de valores, como lo es el conjunto de todos los números reales (números que contienen un punto decimal). Estos dos conjuntos de números, sin embargo, no constituyen un tipo de dato hasta que se incluye un conjunto de operaciones. Estas operaciones, por supuesto, son las operaciones matemáticas y de comparación. La combinación de un conjunto de valores más las operaciones se vuelve un tipo de dato verdadero.

C++ clasifica los tipos de datos en uno de dos agrupamientos fundamentales: tipos de dato de clase y tipos de dato integrados. Un **tipo de dato de clase**, al cual se hace referencia como una clase, para abreviar, es un tipo de dato creado por el programador. Esto significa que el conjunto de valores y operaciones admitidas es definido por un programador, usando código C++.

Un **tipo de dato** integrado es proporcionado como una parte integral del compilador C++ y no requiere código C++ externo. Por tanto, un tipo de dato integrado puede usarse sin recurrir a adiciones de lenguaje complementarias, como las proporcionadas por el archivo de encabezado `iostream` necesario para el objeto `cout`. Los tipos de dato integrados, los cuales también se conocen como tipos **primitivos**, consisten en los tipos numéricos básicos mostrados en la figura 2.7 y las operaciones enumeradas en la tabla 2.2. Como se puede observar en esta tabla, la mayor parte de las operaciones para tipos integrados son proporcionadas como símbolos. Esto contrasta con los tipos de clases, donde la mayor parte de las operaciones son proporcionadas como funciones.

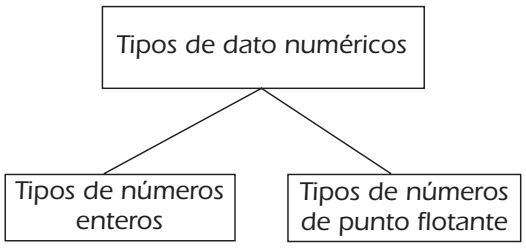


Figura 2.7 Tipos de dato integrados.

Tabla 2.2 Operaciones con tipos de dato integrados

Tipos de dato integrados	Operaciones
`Números enteros`	`+, –, *, /, %, =, ==, !=, <=, >=vsizeof(), y operaciones con bits` (véase el capítulo 15)
`Números de punto flotante`	`+, –, *, /, =, ==, !=, <=, >=, sizeof()`

Al introducir tipos de dato integrados de C++, usaremos literales. Una ***literal*** es un valor aceptable para un tipo de dato. El término ***literal*** refleja que dicho valor se identifica de manera explícita a sí mismo. (Otro nombre para una literal es un ***valor literal*** o ***constante***.) Por ejemplo, todos los números, como 2, 3.6 y –8.2, se denominan valores literales porque despliegan sus valores de manera literal. El texto, como `"¡Hola mundo!"` también se denomina valor literal porque el texto se despliega. Usted ha utilizado valores literales a través de su vida y los ha conocido por lo común como números y palabras. En la sección 2.5, se puede observar algunos ejemplos de valores no literales, es decir, valores que no se despliegan a sí mismos sino que son almacenados y se tiene acceso a ellos por medio de identificadores.

Tipos de datos enteros

C++ proporciona nueve tipos de datos integrados, como se muestra en la figura 2.8. La diferencia esencial entre los diversos tipos de datos enteros es la cantidad de almacenamiento usado por cada tipo, el cual afecta al rango de valores que cada tipo es capaz de representar. Los tres tipos más importantes usados en forma casi exclusiva en la mayor parte de las aplicaciones son los tipos de datos `int`, `char` y `bool`. La razón para los tipos restantes es histórica, ya que fueron implementados para acomodar situaciones especiales (un rango de números pequeño o grande). Esto permitía al programador maximizar el uso de memoria al seleccionar un tipo de datos que usara la cantidad más pequeña de memoria consistente con los requerimientos de una aplicación. Cuando las memorias de las computadoras eran pequeñas y costosas en comparación con las actuales, ésta era una preocupación importante. Aunque ya no es una preocupación para la mayor parte de los programas, estos tipos todavía le proporcionan al programador la capacidad para optimizar el uso de memoria cuando es necesario. Por lo general, estas situaciones ocurren en aplicaciones de ingeniería, como en sistemas de control usados en aparatos domésticos y automóviles.

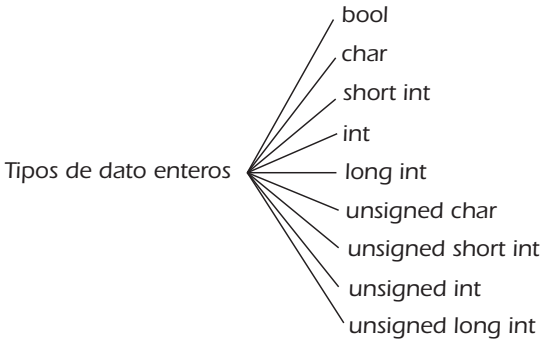


Figura 2.8 *Tipos de dato enteros en C++.*

El tipo de dato `int`

El conjunto de valores admitidos por el tipo de dato `int` son números enteros. Un valor entero consiste sólo de dígitos y de manera opcional puede estar precedido por un signo de más (+) o menos (–). Por tanto, un valor entero puede ser el número cero o cualquier valor numérico positivo o negativo sin un punto decimal. Ejemplos de enteros válidos son:

0 5 –10 +25 1000 253 –26 351 +36

Como ilustran estos ejemplos, los enteros pueden contener un signo explícito. No se permiten comas, puntos decimales ni símbolos especiales, cómo el signo de dólares. Son ejemplos de enteros inválidos:

$255.62 2 523 3. 6 243 892 1 492.89 +6.0

Los diferentes compiladores tienen su propio límite interno para los valores enteros más grande (más positivo) y más pequeño (más negativo) que pueden almacenarse en cada tipo de dato.[6] La asignación de almacenamiento más común es cuatro bytes para el tipo de dato `int`, lo cual restringe el conjunto de valores permitido en este tipo de dato a representar enteros en el rango de `-2,147,483,648` a `2,147,483,647`.[7]

El tipo de dato `char`

El tipo de dato `char` se usa para almacenar caracteres individuales. Los caracteres incluyen las letras del alfabeto (mayúsculas y minúsculas), los diez dígitos 0 a 9 y símbolos especiales como los siguientes: + $. , – y ! Un valor de carácter individual es cualquier letra, dígito o símbolo especial encerrado entre comillas sencillas. Los siguientes son ejemplos de valores de carácter válidos:

```
'A'   '$'   'b'   '7'   'y'   '!'   'M'   'q'
```

[6]Los límites impuestos por el compilador pueden encontrarse en el archivo de encabezado `limits` y se definen como las constantes hexadecimales `int_max` e `int_min`.

[7]En todos los casos, la magnitud del número entero más negativo siempre es un número mayor que la magnitud del entero más positivo. Esto se debe al método de almacenamiento de enteros de complemento a dos, el cual se describió en la sección 1.6.

Punto de Información

Datos atómicos
Un **valor de dato atómico** es un valor considerado una entidad completa en sí misma y que no puede descomponerse en un tipo de dato más pequeño. Por ejemplo, aunque un entero puede descomponerse en dígitos individuales, C++ no posee un tipo de dígito numérico. Más bien, cada entero es considerado como un valor completo por sí mismo y, como tal, se considera un dato atómico. Del mismo modo, debido a que el tipo de dato entero sólo admite valores de datos atómicos, se dice que es un **tipo de dato atómico**. Como se podría esperar, todos los tipos de datos integrados son tipos de datos atómicos.

Los valores de carácter se almacenan por lo general en una computadora usando los códigos ASCII o Unicode. ASCII (se pronuncia as-ki) son las siglas de código estándar americano para intercambio de informacion (American Standard Code for Information Interchange). El código ASCII proporciona códigos para un conjunto de caracteres basado en el idioma inglés, más códigos para control de impresión y de despliegue, como códigos de nueva línea y de expulsión del papel de la impresora. Cada código de carácter está contenido dentro de un solo byte, lo cual proporciona 256 códigos distintos. La tabla 2.3 enumera los códigos en byte ASCII para las letras mayúsculas.

Además, C++ proporciona el código Unicode más nuevo que usa dos bytes por carácter y puede representar 65 536 caracteres. Este código se usa para aplicaciones internacionales al proporcionar conjuntos de caracteres en otros idiomas además del inglés. Como los primeros 256 códigos de Unicode tienen el mismo valor numérico que los 256 códigos ASCII (el byte adicional se codifica con ceros), no es necesario preocuparse por cuál código de almacenamiento se usa cuando se utilizan caracteres en inglés.

Tabla 2.3 Los códigos ASCII para las letras mayúsculas

Letra	Código ASCII	Letra	Código ASCII
A	01000001	N	01001110
B	01000010	O	01001111
C	01000011	P	01010000
D	01000100	Q	01010001
E	01000101	R	01010010
F	01000110	S	01010011
G	01000111	T	01010100
H	01001000	U	01010101
I	01001001	V	01010110
J	01001010	W	01010111
K	01001011	X	01011000
L	01001100	Y	01011001
M	01001101	Z	01011010

Usando la tabla 2.3, podemos determinar cómo se almacenan dentro de una computadora los caracteres ‘B’, ‘A’, ‘R’, ‘T’, ‘E’ y ‘R’, por ejemplo, usando el código de caracteres ASCII. Esta secuencia de seis caracteres requiere seis bytes de almacenamiento (un byte por cada letra) y se almacenaría como se ilustra en la figura 2.9.

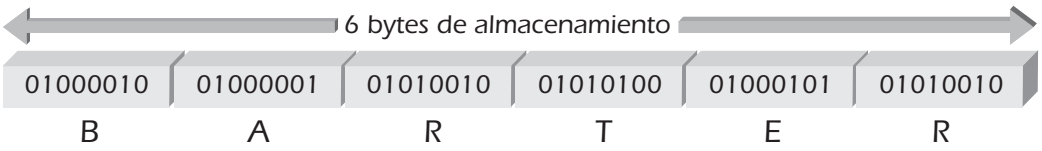


Figura 2.9 *Las letras BARTER almacenadas dentro de una computadora.*

El carácter escape

Un carácter que tiene un significado especial en C++ es la diagonal inversa, \, que se conoce como **carácter de escape**. Cuando se coloca este carácter directamente frente a un grupo selecto de caracteres, esto indica al compilador que escape de la forma en que estos caracteres se interpretarían en forma normal. La combinación de una diagonal inversa y estos caracteres específicos se llama **secuencia de escape**. Se ha mostrado un ejemplo de ésta en la secuencia de escape de línea nueva, `'\n'`, en el capítulo 1. La tabla 2.4 enumera las secuencias de escape más comunes en C++.

Tabla 2.4 Secuencias de escape

Secuencia de escape	Carácter representado	Significado	Código ASCII
`\n`	Línea nueva	Se mueve a una línea nueva	`00001010`
`\t`	Tabulador horizontal	Se mueve a la siguiente posición del tabulador horizontal	`00001001`
`\v`	Tabulador vertical	Se mueve a la siguiente posición del tabulador vertical	`00001011`
`\b`	Retroceso	Retrocede un espacio	`00001000`
`\r`	Retorno de carro	Mueve el cursor al inicio de la línea actual; se escribe para sobrescribir	`00001101`
`\f`	Alimentación de forma	Expulsa una hoja para iniciar otra	`00001100`
`\a`	Alerta	Emite una alerta (por lo general un sonido de campana)	`00000111`
`\\`	Diagonal inversa	Inserta un carácter de diagonal inversa (ésta se usa para colocar un carácter de diagonal inversa real dentro de una cadena)	`01011100`
`\?`	Signo de interrogación	Inserta un carácter de signo de interrogación	`00111111`
`\'`	Comilla sencilla	Inserta un carácter de comilla sencilla (ésta se usa para colocar una comilla sencilla interior dentro de un conjunto de comillas sencillas exteriores)	`00100111`
`\"`	Comillas dobles	Inserta un carácter de comilla doble (ésta se usa para colocar una comilla doble interior dentro de un conjunto de comillas dobles exteriores)	`00100010`
`\`*`nnn`	Número octal	El número *`nnn`* (*`n`* es un dígito) se considerará un número octal	—
`\x`*`hhhh`	Número hexadecimal	El número *`hhhh`* (*`h`* es un dígito) se considerará un número hexadecimal	—
`\0`	Carácter nulo	Inserta el carácter Null, el cual se define con un valor de 0	`00000000`

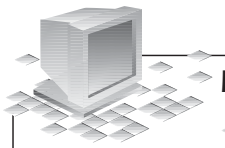
Aunque cada secuencia de escape enumerada en la tabla 2.4 está formada por dos caracteres distintos, la combinación de ambos, sin un espacio en blanco entre ellos, causa que el compilador cree el código individual enumerado en la columna de código ASCII de la tabla 2.4.

El tipo de datos `bool`

En C++, el tipo de datos `bool` se usa para representar datos booleanos (lógicos). Por ello, este tipo de datos está restringido a uno de dos valores: verdadero o falso. Este tipo de datos es más útil cuando un programa debe examinar una condición específica y, debido a la condición de ser verdadera o falsa, tomar un curso de acción prescrito. Por ejemplo, en una aplicación de ventas, la condición que se está examinando podría ser "la compra total es por $100 o más". Sólo cuando esta condición es verdadera se aplica un descuento. Sin embargo, debido a que un tipo de datos booleano usa un código de almacenamiento de enteros, tiene implicaciones útiles explotadas por casi todos los programadores profesionales en C++. Los usos prácticos de las condiciones booleanas se consideran en el capítulo 4, así que pospondremos la exposición de los datos booleanos hasta entonces.

Determinación del tamaño de almacenamiento

Una característica única de C++ es que le permite al programador ver dónde y cómo se almacenan los valores. Por ejemplo, C++ proporciona un operador denominado `sizeof()` que proporciona el número de bytes usados para almacenar valores para cualquier nombre de tipo de datos incluidos dentro de los paréntesis del operador. (Repase la sección 1.6 si no está familiarizado con el concepto de byte.) Éste es un operador integrado que no usa un símbolo aritmético para ejecutar su operación. El programa 2.5 usa este operador para determinar la cantidad de almacenamiento reservado para los tipos de datos, `int`, `char` y `bool`.



Programa 2.5

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

int main()
{
  cout << "\nTipo de datos  Bytes"
       << "\n---------  -----"
       << "\nint          " << sizeof(int)
       << "\nchar         " << sizeof(char)
       << "\nbool         " << sizeof(bool)
       << '\n';

    return 0;
}
```

Punto de Información

El carácter `'\n'` y la cadena `"\n"`

El compilador reconoce tanto `'\n'` como `"\n"` como el carácter línea nueva. La diferencia está en el tipo de datos que se usa. De manera formal, `'\n'` es un carácter literal, mientras `"\n"` es una cadena literal. Desde un punto de vista práctico, ambos hacen que suceda lo mismo: Una línea nueva es forzada en el despliegue de salida. Sin embargo, al encontrar el valor de carácter `'\n'`, el compilador lo traduce usando el código de byte único 00001010 (véase la tabla 2.4). Al encontrar el valor de cadena `"\n"`, el compilador traduce esta cadena usando el código de carácter correcto pero también agrega un carácter extra de fin de cadena, el cual es `'\0'`.

La buena práctica de programación requiere que se termine el último despliegue de salida con una secuencia de escape de línea nueva. Esto asegura que la primera línea de salida de un programa no termina en la última línea desplegada por el programa ejecutado con anterioridad.

Al revisar el programa 2.5, notará que se inserta un solo valor de carácter en `cout` al encerrarlo dentro de comillas sencillas, como en la inserción de la secuencia de escape `'\n'` al final de la instrucción `cout`. Dentro de las primeras cinco líneas desplegadas, este carácter se incluye dentro de cada cadena de salida. Cada vez que el compilador encuentra la secuencia de escape de línea nueva, como un carácter individual o como parte de una cadena, la traduce como un solo carácter que fuerza que el despliegue empiece en una línea nueva. Aunque pueden usarse comillas dobles para la inserción de línea nueva final, como `"\n"`, esto designaría una cadena. Debido a que sólo se transmite un carácter, y para enfatizar que los caracteres únicos se designan usando comillas sencillas, se ha usado `'\n'` en lugar de `"\n"`. Desde un punto de vista práctico, sin embargo, ambas notaciones forzarán una línea nueva en la pantalla.

La salida del programa 2.5 depende del compilador. Es decir, cada compilador reportará en forma correcta la cantidad de almacenamiento que proporciona para el tipo de datos con el que está trabajando. Cuando se ejecutó en la computadora del autor, la cual usa el compilador actual Visual C++.net de Microsoft, se produjo la siguiente salida:

```
Tipos de datos   Bytes
--------------   -----
inf                4
char               1
bool               1
```

Para esta salida, la cual es el almacenamiento típico proporcionado por casi todos los compiladores C++ actuales, podemos determinar el rango de valores que pueden almacenarse en cada uno de estos tipos de datos `int`. Hacerlo, sin embargo, requiere comprender la diferencia entre un tipo de datos con signo y uno sin signo.

Punto de Información

Programas orientados a objetos y orientados a procedimientos

Con excepción del tipo booleano, todos los tipos de datos integrados de C++ son trasladados directo del lenguaje de procedimientos de C. No es sorprendente que los programas que sólo usen tipos integrados individuales no serán programas orientados a objetos. Más bien, como en el programa 2.5, se vuelven programas por procedimientos, es decir, un programa basado primordialmente en procedimientos, como `main()`.

Sólo cuando los tipos integrados se unen para formar un paquete de datos, el cual se convierte en un objeto, puede existir un programa orientado a objetos.

Tipos de datos con signo y sin signo

Un ***tipo de datos con signo*** se define como uno que permite almacenar valores negativos además del cero y valores positivos. Por ello, el tipo de datos `int` es un tipo de datos con signo. Un ***tipo de datos sin signo*** es uno que prevé sólo valores no negativos (es decir, cero y positivos).

Hay casos, sin embargo, donde una aplicación podría requerir sólo valores numéricos sin signo. Por ejemplo, muchas aplicaciones de fecha las almacenan en la forma numérica *añomesdía* (por tanto, la fecha 25/12/2007 se almacenaría como 20071225) y sólo procesan fechas después de 0. Para tales aplicaciones, las cuales nunca requerirán un valor negativo, puede usarse un tipo de datos sin signo.

Todos los tipos enteros sin signo, como `unsigned int`, proporcionan un rango de valores positivos que es, para todos los propósitos prácticos, el doble del rango provisto por su contraparte los enteros con signo. Este rango positivo extra queda disponible al usar el rango negativo de su versión con signo para números positivos adicionales.

Mediante la comprensión de la diferencia entre un tipo de datos con signo y sin signo, la tabla 2.5 puede utilizarse para determinar el rango de valores enteros admitido por los compiladores actuales de C++.

En la tabla 2.5, un `long int` usa la misma cantidad de almacenamiento (cuatro bytes) que un `int`. El único requerimiento del estándar ANSI de C++ es que un `int` debe proporcionar al menos tanto almacenamiento como un `short int`, y un `long int` debe proporcionar al menos tanto almacenamiento como un `int`. En los primeros sistemas de computadora de escritorio (década de los años 80), las cuales estaban limitadas en su capacidad de memoria a miles de bytes, un `short int` usaba de manera típica un byte de almacenamiento, un `int` dos bytes y un `long int` cuatro bytes. Este almacenamiento limitaba el rango de valores `int` de –32 768 a +32 767, mientras el uso de un `unsigned int` proporcionaba un rango de valores de 0 a 65 535, duplicando por tanto el número de valores positivos posibles, lo cual era considerable. Con el rango actual de valores `int` que abarca de –2 a +2 mil millones, la duplicación de valores positivos rara vez es algo que hay que tomar en cuenta. Además, usar un `long int` se vuelve innecesario porque usa la misma capacidad de almacenamiento que un `int`.

Tabla 2.5 Almacenamiento de tipo de datos enteros

Nombre del tipo de datos	Tamaño del **almacenamiento** (en bytes)	Rango de valores
`char`	1	256 caracteres
`bool`	1	verdadero (lo cual es considerado como cualquier valor positivo) y falso (lo cual es un cero)
`short int`	2	–32,768 a +32,767
`unsigned short int`	2	0 a 65,535
`int`	4	–2,147,483,648 a +2,147,483,647
`unsigned int`	4	0 a 4,294,967,295
`long int`	4	–2,147,483,648 a +2,147,483,647
`unsigned long int`	4	0 a 4,294,967,295

Tipos de punto flotante

Un ***número de punto flotante***, al cual se le llama ***número real***, puede ser el número cero o cualquier número positivo o negativo que contenga un punto decimal. Los siguientes son ejemplos de números de punto flotante:

```
+10.625    5.    -6.2    3251.92    0.0    0.33    -6.67    +2.
```

Los números 5., 0.0 y +2. se clasifican como valores de punto flotante, pero los mismos números escritos sin un punto decimal (5, 0, +2) serían valores enteros. Como con los valores enteros, en éstos no se permiten símbolos especiales como el signo de dólar y la coma en los números reales. Los siguientes son ejemplos de números reales inválidos:

```
5,326.25    24    6,459    $10.29    7.007.645
```

C++ acepta tres tipos de datos de punto flotante: `float`, `double` y `long double`. La diferencia entre estos tipos de datos es la cantidad de almacenamiento que usa un compilador para cada tipo. La mayor parte de los compiladores usan el doble de cantidad de almacenamiento para dobles que para flotantes, lo cual permite que un `double` tenga aproximadamente el doble de precisión de un `float`. Por esta razón, a veces se hace referencia a un valor `float` como un número de ***precisión simple*** y a un valor `double` como un número de ***precisión doble***. La asignación de almacenamiento real para cada tipo de datos, sin embargo, depende del compilador particular. El estándar ANSI C++ sólo requiere que un `double` tenga al menos la misma cantidad de precisión que un `float` y un `long double` tenga al menos la misma cantidad de almacenamiento que un `double`. En la actualidad, la mayor parte de los compiladores C++ asignan cuatro bytes para el tipo de datos `float` y ocho bytes para los tipos de datos `double` y `long double`. Esto produce el rango de números mostrado en la tabla 2.6.

Tabla 2.6 Tipos de datos de punto flotante

Tipo	Almacenamiento	Rango absoluto de valores (+ y –)

`float`	`4 bytes`	`1.40129846432481707e-45`
		`a`
		`3.40282346638528860e+38`

`double y`
`long double`
	`8 bytes`	`4.94065645841246544e-324`
		`a`
		`1.79769313486231570e+308`

En compiladores que usan la misma cantidad de almacenamiento para números double y long double, estos dos tipos de datos se vuelven idénticos. (El operador sizeof() que se usó en el programa 2.5 siempre puede usarse para determinar la cantidad de almacenamiento reservada por su compilador para estos tipos de datos.) Una literal tipo `float` se indica al añadir una f o F después del número y se crea un `long double` al añadir una l o L al número. En ausencia de estos sufijos, un número de punto flotante por defecto se establece como `double`. Por ejemplo, observe lo siguiente:

9.234 indica una doble literal
9.234F indica una literal flotante
9.234L indica una literal de doble largo

La única diferencia en estos números es la cantidad de almacenamiento que puede usar una computadora para almacenarlos. El apéndice C describe el formato de almacenamiento binario usado para números de punto flotante y su impacto en la precisión de los números.

Notación exponencial

Los números de punto flotante pueden escribirse en notación exponencial, la cual es semejante a la notación científica y se usa por lo común para expresar valores grandes y pequeños en forma compacta. Los siguientes ejemplos ilustran cómo pueden expresarse números con decimales en notación exponencial y científica:

Notación decimal	Notación exponencial	Notación científica
1625.	1.625e3	1.625×10^3
63421.	6.3421e4	6.3421×10^4
.00731	7.31e-3	7.31 x 10-3
.000625	6.25e-4	6.25 x 10-4

En notación exponencial, la letra e representa exponente. El número que sigue a la e representa una potencia de 10 e indica el número de lugares que debería moverse el punto decimal para obtener el valor decimal estándar. El punto decimal se mueve a la derecha si el número después de la e es positivo o se mueve a la izquierda si el número después de la e es negativo. Por ejemplo, el e3 en 1.625e3 significa mover el punto decimal tres lugares a la derecha, de modo que el número se vuelve 1625. El e–3 en 7.31e–3 significa mover el punto decimal tres lugares a la izquierda de modo que 7.31e–3 se convierte en .00731.

Punto de Información

¿Qué es precisión?

En teoría numérica, el término **precisión** por lo general se refiere a exactitud numérica. En este contexto, se usa una declaración como "este cálculo es exacto, o preciso, hasta el quinto lugar decimal". Esto significa que el quinto dígito después del punto decimal ha sido redondeado y el número es exacto dentro de ±0.00005.

En programación de computadoras, la precisión puede referirse a la exactitud de un número o a la cantidad de dígitos significativos en el número, donde los dígitos significativos se definen como el número de dígitos claramente correctos más 1. Por ejemplo, si el número 12.6874 se ha redondeado hasta el cuarto lugar decimal, es correcto decir que este número es preciso (es decir, exacto) hasta el cuarto lugar decimal. En otras palabras, todos los dígitos en el número son exactos excepto el cuarto dígito decimal, el cual ha sido redondeado. Del mismo modo, este mismo número tiene una precisión de seis dígitos, lo cual significa que los primeros cinco dígitos son correctos y el sexto dígito ha sido redondeado. Otra forma de decir esto es que el número 12.6874 tiene seis dígitos significativos.

Los dígitos significativos en un número no necesitan tener alguna relación con el número de dígitos desplegado. Por ejemplo, si el número 687.45678921 tiene cinco dígitos significativos, sólo es exacto hasta el valor 687.46, donde se supone que el último dígito está redondeado. De una manera similar, los valores en dólares en muchas aplicaciones financieras grandes con frecuencia son redondeados al centenar de miles de dólares más cercano. En tales aplicaciones, un valor en dólares desplegado de $12 400 000, por ejemplo, no es preciso hasta el dólar más cercano. Si este valor se especifica con tres dígitos significativos, sólo es exacto hasta el dígito de cientos de miles.

Ejercicios 2.3

1. Determine los tipos de datos apropiados para los siguientes datos:
   a. el promedio de cuatro calificaciones
   b. el número de días en un mes
   c. la longitud del puente Golden Gate
   d. los números en una lotería estatal
   e. la distancia de Brooklyn, N.Y., a Newark, N.J.
   f. el prefijo de carácter único que especifica un tipo de componente

2. Convierta los siguientes números en forma decimal estándar:

   6.34e5 1.95162e2 8.395e1 2.95e–3 4.623e–4

3. Escriba los siguientes números decimales usando notación exponencial:

   126. 656.23 3426.95 4893.2 .321 .0123 .006789

4. Compile y ejecute el programa 2.5 en su computadora.

5. Modifique el programa 2.5 para determinar el almacenamiento usado por su compilador para todos los tipos de datos enteros de C++.

6. Usando los manuales de referencia del sistema para su computadora, determine el código de caracteres usado por su computadora.

7. Muestre cómo se almacenaría el nombre KINGSLEY dentro de una computadora que usa el código ASCII. Es decir, trace un dibujo similar a la figura 2.9 para el nombre KINGSLEY.

8. Repita el ejercicio 7 usando las letras de su propio apellido.

9. Modifique el programa 2.5 para determinar cuántos bytes asigna su compilador a los tipos de datos `float`, `double` y `long double`.

10. Debido a que las computadoras usan diferentes representaciones para almacenar valores enteros, de punto flotante, de precisión doble y de carácter, analice cómo un programa podría alertar a la computadora de los tipos de datos de los diversos valores que usará.

11. Aunque nos hemos concentrado en operaciones que implican números enteros y de punto flotante, C++ permite que se sumen o resten caracteres y enteros. Esto puede hacerse porque un carácter se almacena usando un código de enteros (es un tipo de datos entero). Por tanto, los caracteres y enteros pueden mezclarse con libertad en expresiones aritméticas. Por ejemplo, si su computadora usa el código ASCII, la expresión 'a' + 1 es igual a 'b', y 'z' – 1 es igual a 'y'. Del mismo modo, 'A' + 1 es 'B' y 'Z' – 1 es 'Y'. Con esto como antecedente, determine el carácter que resulta de las siguientes expresiones. (Asuma que todos los caracteres se almacenan usando el código ASCII.)
    a. 'm' – 5
    b. 'm' + 5
    c. 'G' + 6
    d. 'G' – 6
    e. 'b' – 'a'
    f. 'g' – 'a' + 1
    g. 'G' – 'A' + 1

NOTA: Para completar el siguiente ejercicio, necesita comprender los conceptos básicos del almacenamiento en las computadoras. De manera específica, si no está familiarizado con el concepto de byte, refiérase a la sección 1.6 antes de realizar el siguiente ejercicio.

12. Aunque el número total de bytes varía de una computadora a otra, son comunes los tamaños de memoria de 65 536 a más de varios millones de bytes. En lenguaje de computadora, la letra K representa el número 1 024, el cual es 2 elevado a la décima potencia, y M representa el número 1 048 576, el cual es 2 elevado a la vigésima potencia. Por tanto, un tamaño de memoria de 640K en realidad es 640 por 1024, o 655 360 bytes, y un tamaño de memoria de 4M en realidad es 4 por 1 048 576, lo cual es 4 194 304 bytes. Usando esta información, calcule el número total de bytes en lo siguiente:
    a. una memoria que contiene 128M bytes
    b. una memoria que contiene 256M bytes
    c. una memoria que contiene 512M bytes
    d. una memoria que consta de 256M palabras, donde cada palabra es de 2 bytes
    e. una memoria que consta de 256M palabras, donde cada palabra es de 4 bytes
    f. un disco que especifica 1.44M bytes
    g. un disco que especifica 250MB

2.4 OPERACIONES ARITMÉTICAS

En la sección anterior se presentaron los valores de datos correspondientes para cada tipo de datos integrado de C++. En esta sección, se proporciona el conjunto de operaciones aritméticas que pueden aplicarse a estos valores.

Los números enteros y reales pueden sumarse, restarse, multiplicarse y dividirse. Aunque por lo general es mejor no mezclar números enteros y reales cuando se realizan operaciones aritméticas, se obtienen resultados predecibles cuando se usan diferentes tipos de datos en la misma expresión aritmética. Es sorprendente que se puedan sumar, o restar, datos de carácter a datos de carácter y enteros para producir resultados útiles. (Por ejemplo, 'A' + 1 produce el carácter 'B'.) Esto es posible debido a que los caracteres se almacenan usando códigos de almacenamiento de números enteros.

Los operadores usados en operaciones aritméticas se llaman **operadores aritméticos** y son los siguientes:

Operación	Operador
Adición	+
Sustracción	–
Multiplicación	*
División	/
División de módulo	%

No hay que preocuparse en esta etapa si no entiende el término "división de módulo". Aprenderá más sobre este operador más adelante en esta sección.

Estos operadores se conocen como **operadores binarios**. Este término refleja el hecho que el operador requiere dos operandos para producir un resultado. Un **operando** puede ser un valor literal o un identificador que tiene un valor asociado con él. Una **expresión aritmética binaria simple** consta de un operador aritmético binario que conecta dos valores literales en la forma:

```
valorLiteral operador valorLiteral
```

Los siguientes son ejemplos de expresiones aritméticas binarias simples:

```
3 + 7
18 - 3
12.62 + 9.8
.08 * 12.2
12.6 / 2.
```

Los espacios alrededor de los operadores aritméticos en estos ejemplos se insertan estrictamente para ganar en claridad y pueden omitirse sin afectar el valor de la expresión. Una expresión en C++ debe ser introducida en forma de una línea recta. Por tanto, por ejemplo, la expresión de C++ equivalente a 12.6 dividido entre 2 debe introducirse como 12.6 / 2 y no como la expresión algebraica

$$\frac{12.6}{2}$$

Puede usar `cout` para desplegar el valor de cualquier expresión aritmética en la pantalla de la computadora. Para hacer esto, el valor deseado debe transladarse al objeto. Por ejemplo, esta instrucción produce el despliegue 21:

```
cout << (6 + 15);
```

En sentido estricto, los paréntesis que rodean a la expresión 6 + 15 no se requieren para indicar que el valor de la expresión (es decir, 21) se está desplegando.[8] Además de desplegar un valor numérico, `cout` puede desplegar una cadena identificando la salida, como se hizo en la sección 1.3. Por ejemplo, esta instrucción causa que dos piezas de datos, una cadena y un valor, se envíen a `cout`:

```
cout << "La suma de 6 y 15 es " <<  (6 + 15);
```

De manera individual, cada conjunto de datos enviados a `cout` debe ir precedido por su propio símbolo de inserción de operador (`<<`). Aquí, los primeros datos enviados para despliegue es la cadena `"La suma de 6 y 15 es "`, y el segundo elemento enviado es el valor de la expresión 6 + 15. El despliegue producido por esta instrucción es el siguiente:

```
La suma de 6 y 15 es 21
```

El espacio entre la palabra "es" y el número 21 se da por el espacio colocado dentro de la cadena pasada a `cout`. En lo que respecta a `cout`, su entrada es un conjunto de caracteres que son enviados para ser desplegados en el orden en que son recibidos. Los caracteres de la entrada hacen cola, uno detrás de otro, y son enviados a la consola para su despliegue. Colocar un espacio en la entrada provoca que este espacio sea parte del flujo de caracteres que se despliega. Por ejemplo, examine la siguiente instrucción:

```
cout << "La suma de 12.2 y 15.754 es " <<  (12.2 + 15.754);
```

Que produce el siguiente despliegue:

```
La suma de 12.2 y 15.754 es 27.954
```

Cuando se hacen múltiples inserciones a `cout`, el código puede extenderse a lo largo de múltiples líneas. Sin embargo, sólo debe usarse un punto y coma, el cual se coloca después de la última inserción y termina la instrucción completa. Por tanto, el despliegue anterior es producido por la siguiente instrucción:

```
cout << "La suma de 12.2 y 15.754 es "
     <<  (12.2 + 15.754);
```

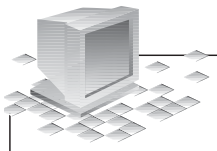
Sin embargo, cuando permite que una instrucción así ocupe múltiples líneas, deben seguirse dos reglas: una cadena contenida entre comillas no puede extenderse más de una línea, y el punto y coma final sólo deberá aparecer en la última línea. Siempre pueden colocarse múltiples símbolos de inserción dentro de una línea.

Los números de punto flotante se despliegan con suficientes lugares decimales a la derecha del punto decimal para acomodar la parte fraccionaria del número. Esto es cierto si el número tiene seis o menos dígitos decimales. Si el número tiene más de seis dígitos decima-

[8]Esto se debe a que el operador + tiene una precedencia mayor que el operador <<; por tanto, la adición se ejecuta antes de la inserción.

les, la parte fraccionaria se redondea a seis dígitos decimales, y si el número no tiene dígitos decimales, no se desplegarán ni un punto decimal ni algún dígito decimal.[9]

El programa 2.6 ilustra el uso de `cout` para desplegar los resultados de expresiones aritméticas dentro de las instrucciones de un programa completo.



Programa 2.6

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

int main()
{
  cout << "15.0 más 2.0 es igual a "            << (15.0 + 2.0) << endl
       << "15.0 menos 2.0 es igual a "          << (15.0 - 2.0) << endl
       << "15.0 por 2.0 es igual a "            << (15.0 * 2.0) << endl
       << "15.0 dividido entre 2.0 es igual a " << (15.0 / 2.0) << endl;

  return 0;
}
```

La salida del programa 2.6 es la siguiente:

```
15.0 más 2.0 es igual a 17
15.0 menos 2.0 es igual a 13
15.0 por 2.0 es igual a 30
15.0 dividido entre 2.0 es igual a 7.5
```

El único elemento nuevo presentado en el programa 2.6 es el término `endl`, el cual es un ejemplo de un manipulador C++. Un **manipulador** es un elemento usado para manipular cómo se despliega el flujo de salida de caracteres. En particular, el manipulador `endl` provoca que primero se inserte un carácter de línea nueva (`'\n'`) en el despliegue y obliga a que todas las inserciones actuales se desplieguen de inmediato, en lugar de esperar por más datos. (La sección 3.2 contiene una lista de los manipuladores más comunes.)

Tipos de expresión

Una **expresión** es cualquier combinación de operadores y operandos que pueden ser evaluados para producir un valor. Una expresión que contiene sólo valores enteros como operandos se llama **expresión entera**, y el resultado de la expresión es un valor entero. Del mismo modo, una expresión que sólo contiene valores de punto flotante (de precisión simple y precisión

[9]No se define ninguna de estas salidas como parte del lenguaje C++. Más bien es definida por un sistema de clases y de rutinas proporcionadas de cada recopilador de C++.

doble) como operandos se llama **expresión de punto flotante**, y el resultado de una expresión así es un valor de punto flotante (también se usa el término **expresión real**). Una expresión que contiene valores enteros y de punto flotante se llama **expresión en modo mixto**. Aunque por lo general es mejor no mezclar valores enteros y de punto flotante en una operación aritmética, el tipo de datos de cada operación se determina por las siguientes reglas:

1. Si ambos operandos son enteros, el resultado de la operación es un entero.
2. Si un operando es un valor real, el resultado de la operación es un valor de precisión doble.

El resultado de una expresión aritmética nunca es un número de precisión simple (`float`). Esto se debe a que, durante la ejecución, un programa C++ convierte de manera temporal todos los números de precisión simple a números de precisión doble cuando se está evaluando una expresión aritmética.

División de enteros

La división de dos valores enteros puede producir resultados bastante extraños para los incautos. Por ejemplo, la expresión 15/2 produce el resultado entero 7. Debido a que los enteros no pueden contener una parte fraccionaria, no puede obtenerse un valor de 7.5. La parte fraccionaria obtenida cuando se dividen dos enteros, es decir, el residuo, siempre se elimina (se trunca). Por tanto, el valor de 9/4 es 2, y 20/3 es 6.

Sin embargo, con frecuencia puede ser necesario conservar el residuo de una división de enteros. Para hacer esto, C++ proporciona un operador aritmético que tiene el símbolo %. Este operador, llamado **módulo** (y también conocido como **operador de residuo**), captura el residuo cuando un número entero es dividido entre un entero (usar un valor no entero con el operador de módulo produce un error en el compilador):

```
 9 % 4 es 1  (es decir, el residuo cuando 9 se divide entre 4 es 1)
17 % 3 es 2  (es decir, el residuo cuando 17 se divide entre 3 es 2)
15 % 4 es 3  (es decir, el residuo cuando 15 se divide entre 4 es 3)
14 % 2 es 0  (es decir, el residuo cuando 14 se divide entre 2 es 0)
```

Con más precisión, el operador de módulo determina primero el número entero de veces que el dividendo, el cual es el número que sigue al operador %, puede dividirse entre el divisor, el cual es el número anterior al operador %. Entonces devuelve el residuo.

Negación

Además de los operadores aritméticos binarios, C++ proporciona operadores unitarios. Un **operador unitario** opera sobre un operando individual. Uno de estos operadores unitario usa el mismo símbolo que la sustracción binaria (–). El signo de menos frente a un valor numérico individual niega el número (invierte su signo).

La tabla 2.7 resume las seis operaciones aritméticas que se han descrito hasta ahora y enumera el tipo de datos para el resultado producido por cada operador, basado en el tipo de datos de los operandos involucrados.

Punto de Información

El manipulador `endl`

En muchos sistemas, el manipulador `endl` y la secuencia de escape `\n` se procesan en la misma forma y producen el mismo efecto. La única excepción se da en aquellos sistemas donde la salida se acumula en forma interna hasta que se colectan suficientes caracteres para que sea ventajoso desplegarlos a todos en una ráfaga en la pantalla. En tales sistemas, los cuales se conocen como sistemas con memoria intermedia o "buffer", el manipulador `endl` fuerza a la salida acumulada a ser desplegada de inmediato, sin esperar que ningún carácter adicional llene el área de buffer antes de imprimirse. Prácticamente, no notará una diferencia en el despliegue final. Por tanto, como regla general, deberá usar la secuencia de escape `\n` siempre que pueda ser incluida dentro de una cadena existente y usar el manipulador `endl` siempre que aparezca `\n` por sí mismo o para determinar de manera formal el final de un grupo específico de despliegue de salida.

Tabla 2.7 Resumen de operadores aritméticos

Operación	Operador	Tipo	Operando	Resultado
Adición	+	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Sustracción	-	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Multiplicación	*	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
División	/	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Módulo	%	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Negación	-	Unitario	Entero o doble	Igual que el operando

Precedencia del operador y asociatividad

Además de expresiones simples como 5 + 12 y .08 * 26.2, pueden crearse expresiones aritméticas más complejas. C++, como la mayor parte de otros lenguajes de programación, re-

quiere que se sigan ciertas reglas cuando se escriben expresiones que contengan más de un operador aritmético. Estas reglas son las siguientes:

1. Nunca deben colocarse dos símbolos de operadores aritméticos binarios uno al lado del otro. Por ejemplo, 5 * % 6 es inválido porque los dos operadores, * y %, están colocados uno junto al otro.
2. Pueden usarse paréntesis para formar agrupamientos, y todas las expresiones encerradas dentro de paréntesis son evaluadas primero. Esto permite a los paréntesis alterar la evaluación en cualquier orden deseado. Por ejemplo, en la expresión (6 + 4) / (2 + 3), 6 + 4 y 2 + 3 se evalúan primero para producir 10 / 5. Luego se evalúa 10 / 5 para producir 2.
3. Conjuntos de paréntesis pueden ser encerrados por otros paréntesis. Por ejemplo, la expresión (2 * (3 + 7)) / 5 es válida y evalúa para 4. Cuando se incluyen paréntesis dentro de paréntesis, las expresiones en los paréntesis interiores siempre se evalúan primero. La evaluación continúa desde los paréntesis más interiores hasta los más exteriores hasta que se han evaluado las expresiones en todos los paréntesis. El número de paréntesis de cierre,), siempre debe ser igual al número de paréntesis de apertura, (, de modo que no existan conjuntos sin par.
4. No pueden usarse paréntesis para indicar multiplicación; en su lugar, debe usarse el operador de multiplicación, *. Por ejemplo, la expresión (3 + 4) (5 + 1) es inválida. La expresión correcta es (3 + 4) * (5 + 1).

Los paréntesis deberían especificar agrupamientos lógicos de operandos e indicarle con claridad, al compilador y a los programadores, el orden previsto de las operaciones aritméticas. Aunque las expresiones dentro de paréntesis siempre se evalúan primero, las expresiones que contienen múltiples operadores, con y sin paréntesis, se evalúan por la prioridad, o **precedencia**, de los operadores. Hay tres niveles de precedencia:

- P1: todas las negaciones se realizan primero.
- P2: a continuación se calculan las operaciones de multiplicación, división y módulo. Las expresiones que contienen más de un operador de multiplicación, división o módulo se evalúan de izquierda a derecha conforme se encuentra cada operador. Por ejemplo, en la expresión 35 / 7 % 3 * 4, todas las operaciones son de la misma prioridad, así que las operaciones se ejecutarán de izquierda a derecha conforme se encuentre cada operador. Por tanto, la división se hace primero, produciendo la expresión 5 % 3 * 4. La operación de módulo se ejecuta a continuación, produciendo un resultado de 2. Por último, se calcula el valor de 2 * 4 para producir 8.
- P3: la adición y la sustracción se calculan al último. Las expresiones que contienen más de una adición o sustracción se evalúan de izquierda a derecha conforme se encuentre cada operador.

Además de la precedencia, los operadores tienen una ***asociatividad***, la cual es el orden en que se evalúan los operadores de la misma precedencia, como se describió en la regla P2. Por ejemplo, ¿la expresión 6.0 * 6/4 produce 9.0, lo cual es (6.0 * 6)/4: o 6.0, lo cual es 6.0 * (6/4)? La respuesta es 9.0, porque los operadores de C++ usan la misma asociatividad que en

las matemáticas generales, las cuales evalúan la multiplicación de izquierda a derecha, como lo indica la regla P2. La tabla 2.8 enumera la precedencia y asociatividad de los operadores considerados en esta sección. Como se ha visto, la precedencia de un operador establece su prioridad en relación con todos los demás operadores. Los operadores en la parte superior de la tabla 2.8 tienen una prioridad mayor que los operadores en la parte inferior de la tabla. En expresiones que contienen múltiples operadores de diferente precedencia, el operador con la mayor precedencia se usa antes que un operador con menor precedencia. Por ejemplo, en la expresión 6 + 4 / 2 + 3, debido a que el operador de división tiene una precedencia mayor (P2) que la adición, la división se hace primero, produciendo un resultado intermedio de 6 + 2 + 3. Entonces se ejecutan las adiciones, de izquierda a derecha, para producir un resultado final de 11.

Tabla 2.8 Precedencia y asociatividad de operadores

Operador	Asociatividad
unitario –	derecha a izquierda
* / %	izquierda a derecha
+ –	izquierda a derecha

Por último, usaremos la tabla 2.8 o las reglas de precedencia para evaluar una expresión que contiene operadores de diferente precedencia, como 8 + 5 * 7 % 2 * 4. Debido a que los operadores de multiplicación y módulo tienen una mayor precedencia que el operador de adición, estas dos operaciones se evalúan primero (P2), usando su asociatividad de izquierda a derecha, antes que se evalúe la adición (P3). Por tanto, la expresión completa se evalúa como sigue:

```
8 + 5 * 7 % 2 * 4 =
   8 + 35 % 2 * 4 =
        8 + 1 * 4 =
            8 + 4 = 12
```

Ejercicios 2.4

1. A continuación se enumeran expresiones algebraicas correctas y expresiones C++ incorrectas correspondientes a ellas. Encuentre los errores y escriba las expresiones C++ corregidas.

	Álgebra	*Expresión C++*
a.	`(2)(3) + (4)(5)`	`(2)(3) + (4)(5)`

b.	`6 + 18`
	`------`
	`  2`
		`6 + 18 / 2`

c.	`  4.5`
	`----------`
	`12.2 - 3.1`
		`4.5 / 12.2 - 3.1`

d. `4.6(3.0 + 14.9)` `4.6(3.0 + 14.9)`

e. `(12.1 + 18.9)(15.3 - 3.8)` `(12.1 + 18.9)(15.3 - 3.8)`

2. Determine el valor de las siguientes expresiones enteras:

a. 3 + 4 * 6
b. 3 * 4 / 6 + 6
c. 2 * 3 / 12 * 8 / 4
d. 10 * (1 + 7 * 3)
e. 20 – 2 / 6 + 3
f. 20 – 2 / (6 + 3)
g. (20 – 2) / 6 + 3
h. (20 – 2) / (6 + 3)
i. 50 % 20
j. (10 + 3) % 4

3. Determine el valor de las siguientes expresiones de punto flotante:

a. 3.0 + 4.0 * 6.0
b. 3.0 * 4.0 / 6.0 + 6.0
c. 2.0 * 3.0 / 12.0 * 8.0 / 4.0
d. 10.0 * (1.0 + 7.0 * 3.0)
e. 20.0 – 2.0 / 6.0 + 3.0
f. 20.0 – 2.0 / (6.0 + 3.0)
g. (20.0 – 2.0) / 6.0 + 3.0
h. (20.0 – 2.0) / (6.0 + 3.0)

4. Evalúe las siguientes expresiones en modo mixto y enumere el tipo de datos del resultado. Al evaluar las expresiones, esté consciente de los tipos de datos de todos los cálculos intermedios.

a. 10.0 + 15 / 2 + 4.3
b. 10.0 + 15.0 / 2 + 4.3
c. 3.0 * 4 / 6 + 6
d. 3 * 4.0 / 6 + 6
e. 20.0 – 2 / 6 + 3
f. 10 + 17 * 3 + 4
g. 10 + 17 / 3. + 4
h. 3.0 * 4 % 6 + 6
i. 10 + 17 % 3 + 4.

5. Suponga que `amount` almacena el valor entero 1, `m` almacena el valor entero 50, `n` almacena el valor entero 10 y `p` almacena el valor entero 5. Evalúe las siguientes expresiones:

a. `n / p + 3`
b. `m / p + n - 10 * amount`
c. `m - 3 * n + 4 * amount`
d. `amount / 5`
e. `18 / p`
f. `-p * n`
g. `-m / 20`
h. `(m + n) / (p + amount)`
i. `m + n / p + amount`

6. Repita el ejercicio 5, suponiendo que `amount` almacena el valor 1.0, `m` almacena el valor 50.0, `n` almacena el valor 10.0 y `p` almacena el valor 5.0.
7. Introduzca, compile y ejecute el programa 2.2 en su sistema de cómputo.
8. Determine la salida del siguiente programa:

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

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

  return 0;
}
```

9. 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;
  cout << "\nEl residuo de 17 dividido entre 3 es " << 17 % 3;

  return 0;
}
```

10. Escriba un programa 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.
11. Escriba un programa 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 los valores desplegados son correctos.

2.5 VARIABLES E INSTRUCCIONES DE DECLARACIÓN

Todos los valores enteros, de punto flotante y otros usados en un programa de computadora se almacenan en la unidad de memoria de la computadora y se recuperan de ella. Desde el punto de vista conceptual, las ubicaciones individuales de memoria en la unidad de memoria están ordenadas como las habitaciones en un gran hotel. Como en un hotel, cada ubicación de memoria tiene una dirección única ("número de habitación"). Antes que existieran lenguajes de alto nivel como C++, se hacía referencia a las ubicaciones de memoria por sus direcciones. Por ejemplo, almacenar los valores enteros 45 y 12 en las ubicaciones de memoria 1652 y 2548 (véase la figura 2.10), respectivamente, requería instrucciones equivalentes a

Coloque un 45 en la ubicación 1652
Coloque un 12 en la ubicación 2548

Para sumar los dos números que se acaban de almacenar y guardar el resultado en otra ubicación de memoria, por ejemplo en la ubicación 3000, se necesita una instrucción comparable a

Sume el contenido de la ubicación 1652
al contenido de la ubicación 2548
y almacene el resultado en la ubicación 3000

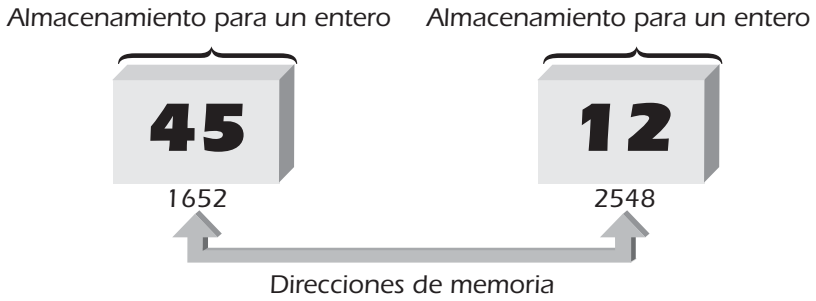


Figura 2.10 *Almacenamiento suficiente para dos enteros.*

Es evidente que este método de almacenamiento y recuperación es un proceso engorroso. En lenguajes de alto nivel como C++, se usan nombres simbólicos en lugar de direcciones de memoria reales. Estos nombres simbólicos se llaman **variables**. Una variable es tan sólo un nombre dado por el programador para referirse a ubicaciones de almacenamiento de la computadora. Se usa el término variable porque el valor almacenado en la variable puede cambiar, o variar. Para cada nombre que usa el programador, la computadora se mantiene al tanto de la dirección de memoria real correspondiente a ese nombre. En nuestra analogía de la habitación de hotel, esto es equivalente a poner un nombre en la puerta de una habitación y referirse a la habitación con ese nombre, como la habitación AZUL, en lugar de usar el número real de la habitación.

En C++, la selección de nombres de variables se deja al programador siempre que observe las reglas para seleccionar nombres de identificadores. Éstos se presentaron en la página 35 y se resumen a continuación.

1. El nombre de la variable debe comenzar con una letra o subrayado (_) y sólo puede contener letras, subrayados o dígitos. No puede contener ningún espacio en blanco, comas ni símbolos especiales, como ( ) & , $ # . ! \ ?
2. Un nombre de variable no puede ser una palabra clave (véase la tabla 2.1).
3. El nombre de la variable no puede consistir en más de 1024 caracteres.

Además, los nombres de variables deben ser mnemónicos que den algún indicio del uso de la variable. Por ejemplo, un buen nombre para una variable usada para almacenar un valor que es el total de algunos otros valores sería `suma` o `total`. No deberán seleccionarse nombres de variables que no dan ningún indicio del valor almacenado, como `r2d2`, `linda`, `beto` y `getum`. Como con los nombres de función, los nombres de variables pueden escribirse con letras mayúsculas y minúsculas.

Ahora suponga que a la primera ubicación de memoria ilustrada en la figura 2.11, la cual tiene la dirección 1652, se le da el nombre `num1`. También suponga que a la ubicación de memoria 2548 se le da el nombre de variable `num2`, y a la ubicación de memoria 3000 se le da el nombre de variable `total`, como se ilustra en la figura 2.11.

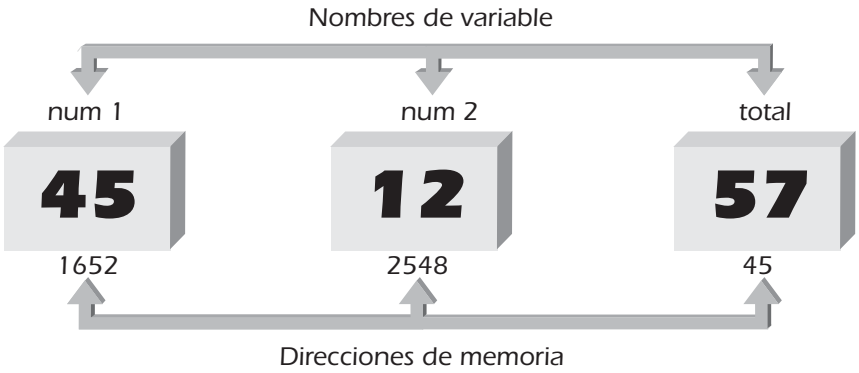


Figura 2.11 *Denominación de las ubicaciones de almacenamiento.*

Usando estos nombres de variable, la operación de almacenar 45 en la ubicación 1652, almacenar 12 en la ubicación 2548 y sumar los contenidos de estas dos ubicaciones se logra con las instrucciones de C++

```
num1 = 45;
num2 = 12;
total = num1 + num2;
```

Cada una de estas tres instrucciones se llama **instrucción de asignación** porque le indica a la computadora que asigne (almacene) un valor en una variable. Las instrucciones de asignación siempre tienen un signo de igual (=) y un nombre de variable inmediatamente a la izquierda de este signo. El valor a la derecha del signo de igual se determina primero, y este valor se asigna a la variable a la izquierda del signo de igual. Los espacios en blanco en las instrucciones de asignación se insertan para una mayor legibilidad. En el siguiente capítulo se hablará más sobre las instrucciones de asignación, pero por ahora pueden usarse para almacenar valores en variables.

Un nombre de variable es de utilidad porque libera al programador en lo que concierne a dónde se almacenan los datos en forma física dentro de la computadora. Tan sólo se usa el nombre de variable y se deja que el compilador se preocupe por el lugar de la memoria en que se almacenan los datos. Sin embargo, antes de almacenar un valor en una variable, C++ requiere que se declare con claridad el tipo de datos que se van a almacenar en ella. Debe indicarse al compilador, con anticipación, los nombres de las variables que se usarán para caracteres, los nombres que se usarán para enteros y los nombres que se usarán para almacenar los otros tipos de datos de C++.

Instrucciones de declaración

Nombrar una variable y especificar el tipo de datos que pueden almacenarse en ella se logra usando **instrucciones de declaración**. Una instrucción de declaración tiene la forma general

tipo-de-datos nombreDeVariable;

Punto de Información

Datos atómicos
Las variables que declaramos aquí se han usado para almacenar valores de datos atómicos. Un valor de datos atómicos es aquel que se considera una entidad completa por sí misma y que no puede descomponerse en un tipo de datos más pequeño respaldado por el lenguaje. Por ejemplo, aunque un entero puede descomponerse en dígitos individuales, C++ posee un tipo de dígitos numéricos. Más bien, cada entero es considerado como un valor completo por sí mismo y, como tal, se considera dato atómico. Del mismo modo, en vista que el tipo de datos enteros sólo respalda valores de datos atómicos, se dice que es un tipo de datos atómicos. Como podría esperarse, double, char y bool también son tipos de datos atómicos.

donde *tipo-de-datos* designa un tipo de datos válidos en C++ y *nombreDeVariable* es un nombre de variable seleccionado por el usuario. Por ejemplo, las variables usadas para contener valores enteros se declaran usando la palabra clave `int` para especificar el tipo de datos y tiene la forma:

`int` *nombreDeVariable;*

Por tanto, la instrucción de declaración

```
int sum;
```

declara `suma` como el nombre de una variable capaz de almacenar un valor entero.

Además de la palabra reservada `int` usada para especificar un entero, la palabra reservada ***long*** se usa para especificar un entero largo.[10] Por ejemplo, la instrucción

```
long fechanum;
```

declara `fechanum` como una variable que se usará para almacenar un entero largo. Cuando se usa el calificador `long` puede incluirse la palabra clave `int`. Por tanto, la declaración anterior también puede escribirse como

```
long int fechanum;
```

Las variables usadas para contener valores de precisión simple se declaran usando la palabra clave `float`, mientras las variables que se usarán para contener valores de precisión doble se declaran usando la palabra clave `double`. Por ejemplo, la instrucción

```
float primernum;
```

declara `primernum` como una variable que se usará para almacenar un número de precisión simple. Del mismo modo, la instrucción

```
double segundonum;
```

declara que la variable `segundonum` se usará para almacenar un número de precisión doble.

Aunque las instrucciones de declaración pueden colocarse en cualquier parte dentro de una función, por lo general las declaraciones se agrupan y se colocan inmediatamente después de la llave de apertura de la función. En todos los casos, sin embargo, una variable de-

[10]Además, las palabras reservadas `unsigned int` se usan para especificar un entero que sólo puede almacenar números no negativos y la palabra reservada `short` se usa para especificar un entero corto.

be declararse antes que pueda usarse y, como todas las instrucciones de C++, las instrucciones de declaración deben terminar con un punto y coma. Si las instrucciones de declaración se colocan después de la llave de apertura de la función, una función `main()` simple que contenga instrucciones de declaración tendría la forma general

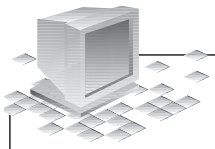
```
#include <iostream>
using namespace std;

int main()
{
  instrucciones de declaración;

  otras instrucciones;

  return 0;
}
```

El programa 2.7 ilustra esta forma al declarar y usar cuatro variables de precisión doble, usando el objeto `cout` para desplegar el contenido de una de las variables.



Programa 2.7

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

int main()
{
  double calif1; // declara calif1 como una variable de precisión doble
  double calif2; // declara calif2 como una variable de precisión doble
  double total;   // declara total como una variable de precisión doble
  double promedio; // declara promedio como una variable de precisión doble

  calif1 = 85.5;
  calif2 = 97.0;
  total = calif1 + calif2;
  promedio = total/2.0;  // divide el total entre 2.0
  cout << "El promedio de las calificaciones es " << promedio << endl;

  return 0;
}
```

La colocación de las instrucciones de declaración en el programa 2.7 es simple, aunque pronto se verá que las cuatro declaraciones individuales pueden combinarse en una sola declaración. Cuando se ejecuta el programa 2.7, se despliega la siguiente salida:

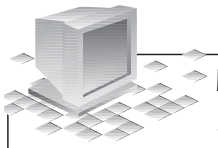
```
El promedio de las calificaciones es 91.25
```

Hay que observar que cuando se inserta un nombre de variable en un objeto `cout`, el valor almacenado en la variable se coloca en el flujo de salida y se despliega.

Del mismo modo en que las variables de números enteros y reales (de precisión simple, de precisión doble y de doble largo) deben declararse antes que puedan utilizarse, también debe declararse una variable usada para almacenar un carácter individual. Las variables de carácter se declaran usando la palabra reservada `char`. Por ejemplo, la declaración

```
char ch;
```

declara que `ch` es una variable de carácter. El programa 2.8 ilustra esta declaración y el uso de `cout` para desplegar el valor almacenado en una variable de carácter.



Programa 2.8

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

int main()
{
  char ch;     // esto declara una variable de carácter

  ch = 'a';    // almacena la letra a en ch
  cout << "El carácter almacenado en ch es " << ch << endl;
  ch = 'm';    // ahora almacena la letra m en ch
  cout << "El carácter almacenado ahora en ch es "<< ch << endl;

  return 0;
}
```

Cuando se ejecuta el programa 2.8, la salida producida es:

```
El carácter almacenado en ch es a
El carácter almacenado ahora en ch es m
```

Hay que observar en el programa 2.8 que la primera letra almacenada en la variable `ch` es a y la segunda letra almacenada es m. En vista que una variable sólo puede usarse para almacenar un valor a la vez, la asignación de m a la variable causa de manera automática que a se sobrescriba.

Declaraciones múltiples

Las variables que tienen el mismo tipo de datos siempre pueden agruparse y declararse usando una sola instrucción de declaración. La forma común de dicha declaración es

tipo de datos listaDeVariables;

Por ejemplo, las cuatro declaraciones separadas usadas en el programa 2.7,

```
double calif1;
double calif2;
double total;
double promedio;
```

pueden reemplazarse por la instrucción de declaración sencilla

```
double calif1, calif2, total, promedio;
```

Del mismo modo, las dos declaraciones de carácter,

```
char ch;
char clave;
```

pueden reemplazarse con la instrucción de declaración sencilla

```
char ch, clave;
```

Hay que observar que declarar múltiples variables en una sola declaración requiere que el tipo de datos de las variables sólo se dé una vez, que todos los nombres de las variables se separen con comas y que sólo se use un punto y coma para terminar la declaración. El espacio después de cada coma se inserta por legibilidad y no es indipensable.

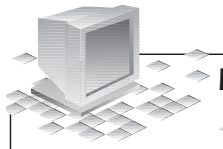
Las instrucciones de declaración también pueden usarse para almacenar un valor en variables declaradas. Por ejemplo, la instrucción de declaración

```
int num1 = 15;
```

declara al mismo tiempo la variable `num1` como una variable entera y establece el valor de 15 en la variable. Cuando se usa una instrucción de declaración para almacenar un valor en una variable, se dice que la variable fue ***inicializada***. Por tanto, en este ejemplo es correcto decir que la variable `num1` fue inicializada en 15. Del mismo modo, las instrucciones de declaración

```
double calif1 = 87.0;
double calif2 = 93.5;
double total;
```

declaran tres variables de precisión doble e inicializan dos de ellas. Cuando se usan inicializaciones, la buena práctica de programación dicta que cada variable inicializada sea declarada en una línea individual. Pueden usarse como inicializadores dentro de una función las constantes, las expresiones que usan sólo constantes (como 87.0 + 12 – 2) y expresiones que usan constantes y variables inicializadas con anterioridad. Por ejemplo, el programa 2.7 con una inicialización de declaración se convierte en el programa 2.7a.



Programa 2.7a

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

int main()
{
  double calif1 = 85.5;
  double calif2 = 97.0;
  double total, promedio;

  total = calif1 + calif2;
  promedio = total/2.0;  // divide el total entre 2.0
  cout << "El promedio de las calificaciones es " << promedio << endl;

  return 0;
}
```

Observe la línea en blanco después de la última instrucción de declaración. Insertar una línea en blanco después de las declaraciones de variables colocadas en la parte superior del cuerpo de una función es una buena práctica de programación. Mejora tanto la apariencia de un programa como su legibilidad.

Una característica interesante de C++ es que las declaraciones de variables pueden entremezclarse libremente e incluso estar contenidas con otras instrucciones; el único requisito es que una variable debe declararse antes de usarla. Por ejemplo, la variable `total` en el programa 2.7a podría haberse declarado cuando se usó por primera vez empleando la instrucción double `total = calif1 + calif2`. En situaciones muy restringidas (como en la depuración, como se describe en la sección 3.9, o en una iteración tipo for, descrita en la sección 5.4), puede ser útil declarar una variable en el punto de su primer uso. En general, sin embargo, es preferible no dispersar las declaraciones sino más bien agruparlas en la manera más concisa y clara posible, al principio de cada función.

Asignación de memoria

Las instrucciones de declaración que se han introducido ejecutan tareas tanto de software como de hardware. Desde una perspectiva del software, las instrucciones de declaración siempre proporcionan una lista de todas las variables y sus tipos de datos. En esta función de software, las declaraciones de variable también ayudan a controlar un error común y problemático causado por la escritura equivocada del nombre de una variable dentro de un programa. Por ejemplo, suponga que una variable nombrada `distancia` se declara e inicializa usando la instrucción

```
int distancia = 26;
```

Ahora suponga que esta variable se escribe mal en forma inadvertida en la instrucción

```
mpg = distncia / galones;
```

En lenguajes que no requieren declaraciones de variable, el programa trataría `distancia` como una variable nueva y le asignaría un valor inicial de cero a la variable o usaría cualquier valor que resultara estar en el área de almacenamiento de la variable. En cualquier caso se calcularía un valor y se asignaría a `mpg`, y encontrar el error o siquiera saber que ocurrió podría ser en extremo problemático. Tales errores son imposibles en C++ porque el compilador señalará `distancia` como una variable no declarada. El compilador no puede, por supuesto, detectar cuándo una variable declarada se escribe en lugar de otra variable declarada.

Además de su función en el software, las instrucciones de declaración también pueden realizar una tarea de hardware distinta. Debido a que cada tipo de datos tiene sus propios requerimientos de almacenamiento, la computadora puede asignar suficiente almacenamiento para una variable sólo después de conocer el tipo de datos de ésta. Debido a que las declaraciones de variable proporcionan esta información, pueden usarse para forzar al compilador a reservar almacenamiento suficiente en la memoria física para cada variable. Las instrucciones de declaración usadas con este propósito de hardware se llaman también ***instrucciones de definición*** porque definen o le indican al compilador cuánta memoria es necesaria para el almacenamiento de datos.

Todas las instrucciones de declaración que hemos encontrado hasta ahora han sido instrucciones de definición. Más adelante, se verán casos de instrucciones de declaración que no causan que se asigne ningún almacenamiento nuevo y se usan tan sólo para declarar o alertar al programa de los tipos de datos de variables que se crean en otras partes del programa.

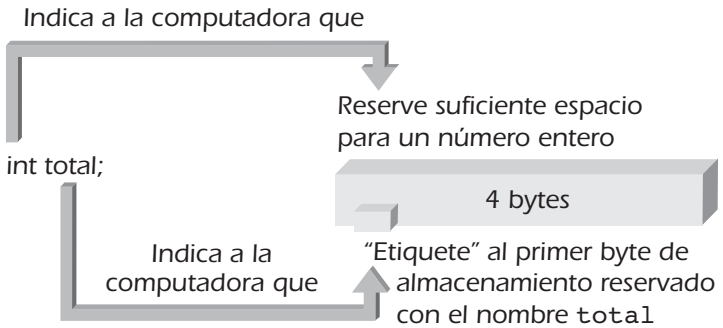


Figura 2.12a *Definición de una variable de entero llamada* `total`.

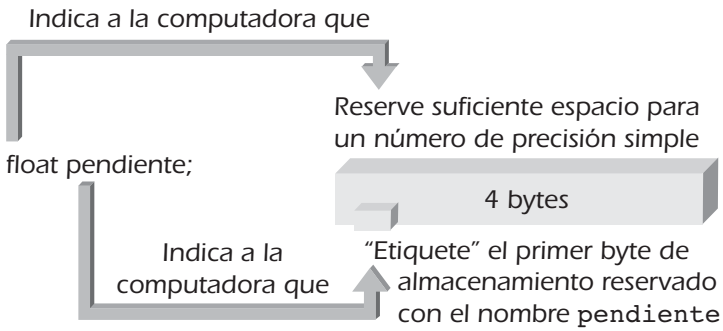


Figura 2.12b *Definición de la variable de punto flotante llamada* `pendiente`.

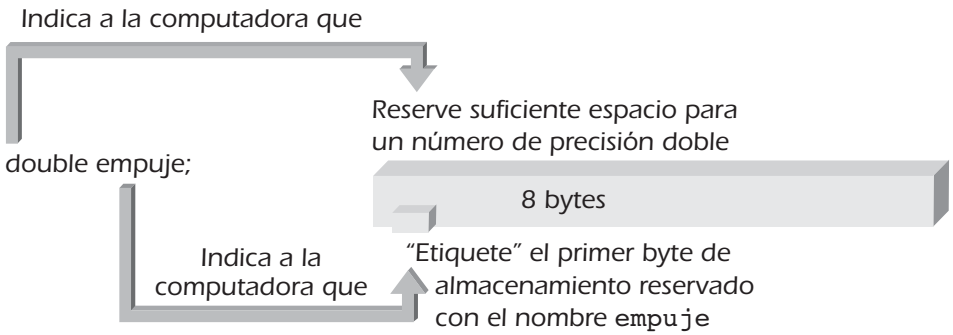


Figura 2.12c *Definición de la variable de precisión doble llamada* `empuje`.

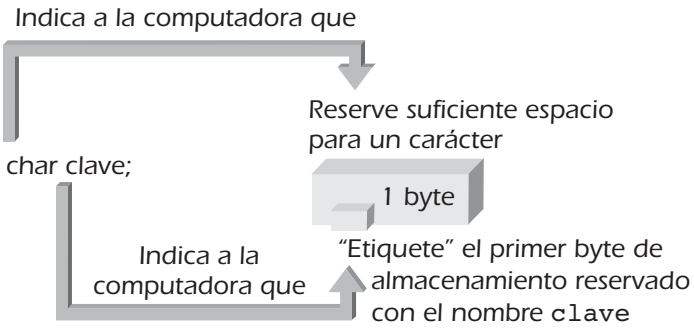


Figura 2.12d *Definición de la variable de carácter llamada* `clave`.

La figura 2.12 (partes a-d) ilustra la serie de operaciones puestas en movimiento por instrucciones de declaración que también desempeñan un papel de definición. La figura muestra que las instrucciones de definición (o, si lo prefiere, instrucciones de declaración que también causan que se asigne memoria) "etiquetan" el primer byte de cada conjunto de bytes reservados con un nombre. Este nombre es, por supuesto, el nombre de la variable y es usado por la computadora para ubicar en forma correcta el punto de inicio del área de memoria reservada de cada variable.

Dentro de un programa, después que se ha declarado una variable, por lo general ésta es usada por el programador para referirse al contenido de la variable (es decir, el valor de la variable). En qué parte de la memoria se almacena este valor comúnmente es de poco interés para el programador. El compilador, sin embargo, debe estar al tanto de dónde se almacena cada valor y ubicar en forma correcta cada variable. En esta tarea la computadora usa el nombre de la variable para ubicar el primer byte de almacenamiento asignado con anterioridad a la variable. Conocer el tipo de datos de la variable le permite luego al compilador almacenar o recuperar el número correcto de bytes.

Despliegue de la dirección de una variable[11]

Cada variable tiene tres elementos importantes asociados a ella: su tipo de datos, el valor real almacenado en la variable y la dirección de la variable. El valor almacenado en la variable se conoce como el contenido de la variable, mientras la dirección de la primera ubicación de

[11]Este tema puede omitirse en la primera lectura sin perder la continuidad.

memoria usada para la variable constituye su dirección. Cuántas ubicaciones se usan en realidad para la variable, como acabamos de ver, depende del tipo de datos de ésta. La relación entre estos tres elementos (tipo, contenido, ubicación) se ilustra en la figura 2.13.

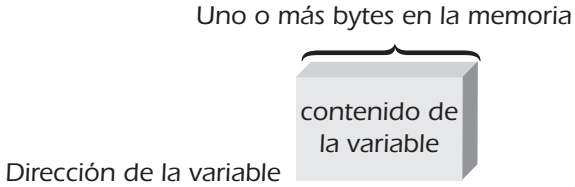


Figura 2.13 *Una variable típica.*

Los programadores por lo general sólo se interesan en el valor asignado a la variable (su contenido) y ponen poca atención al lugar donde está almacenado el valor (su dirección). Por ejemplo, considere el programa 2.9.

Programa 2.9

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

int main()
{
  int num;

  num = 22;
  cout << "El valor almacenado en num es " << num << endl;

  return 0;
}
```

La salida desplegada cuando se ejecuta el programa 2.9 es

```
El valor almacenado en num es 22
```

El programa 2.9 tan sólo imprime el valor 22, el cual es el contenido de la variable `num`. Podemos ir más allá, sin embargo, y preguntar "¿Dónde está almacenado en realidad el número 22?" Aunque la respuesta es "en `num`", ésta es sólo la mitad de la respuesta. El nombre de la variable `num` tan sólo es un símbolo conveniente para ubicaciones físicas reales en la memoria, como se ilustra en la figura 2.14.

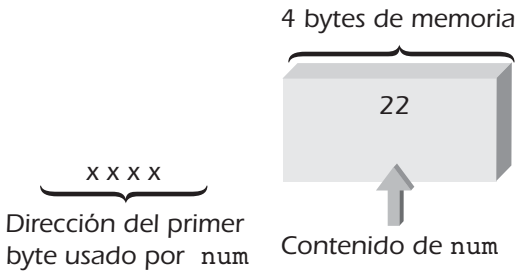
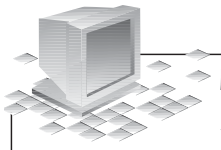


Figura 2.14 *Algún lugar en la memoria.*

Para determinar la dirección de `num`, podemos usar el operador de dirección de C++, `&`, el cual significa "la dirección de". Excepto cuando se usa en una expresión, el operador de dirección colocado enfrente del nombre de una variable se refiere a la dirección de la variable.[12] Por ejemplo, `&num` significa *la dirección de* `num`, `&total` significa *la dirección de* `total` y `&precio` significa *la dirección de* `precio`. El programa 2.10 usa el operador de dirección para mostrar la dirección de la variable `num`.



Programa 2.10

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

int main()
{
  int num;

  num = 22;
  cout << "El valor almacenado en num es " << num << endl;
  cout << "La dirección de num = " << &num << endl;

  return 0;
}
```

La salida del programa 2.10 es

```
El valor almacenado en num es 22
La dirección de num = 0012FED4
```

[12]Cuando se usa para declarar variables y argumentos de referencia, los cuales se presentan en el capítulo 7, el signo & se refiere al tipo de datos que lo *preceden*. Por tanto, la declaración `double &num` se lee como "`num` es la dirección de un `double`" o, de manera más común, como "`num` es una referencia a un `double`"

La figura 2.15 ilustra la información de dirección adicional proporcionada por la salida del programa 2.10.

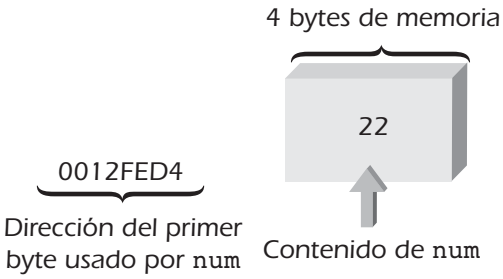


Figura 2.15 *Un panorama más completo de la variable* `num`.

Es evidente que la salida de dirección del programa 2.10 depende de la computadora usada para ejecutar el programa. Sin embargo, cada vez que se ejecuta el programa 2.10 muestra la dirección de la primera ubicación de memoria usada para almacenar la variable `num`. Como lo ilustra la salida del programa 2.10, el despliegue de la dirección está en notación hexadecimal. Este despliegue no tiene efecto en la manera en que se usan las direcciones en forma interna en el programa; tan sólo nos proporciona un medio de desplegar direcciones que es útil para comprenderlas. Como se verá en los capítulos 6 y 12, usar direcciones, en oposición a sólo desplegarlas, es una herramienta de programación importante y poderosa en extremo.

Ejercicios 2.5

1. Establezca si los siguientes nombres de variables son válidos o no. Si son inválidos, explique por qué.

```
prod_a      c1234       abcd        _c3         12345
ampnuevo    vatios      $total      $alnuevo    a1b2c3d4
9ab6        suma.de     promedio    voltios1    finvoltios
```

2. Establezca si los siguientes nombres de variable son válidos o no. Si son inválidos, explique por qué. También indique cuál de los nombres de variable válidos no deberían usarse debido a que no transmiten información sobre la variable.

```
actual      a243        r2d2        primer_num      cc_a1
hector      susana      c3p0        total           suma
maximo      okay        a           increible       veporel
3suma       for         tot.a1      c$cinco         potencianeta
```

3. **a.** Escriba una instrucción de declaración para declarar que se usará la variable `cuenta` para almacenar un entero.
   b. Escriba una instrucción de declaración para declarar que se usará la variable `voltio` para almacenar un número de punto flotante.
   c. Escriba una instrucción de declaración para declarar que se usará la variable `potencia` para almacenar un número de precisión doble.

d. Escriba una instrucción de declaración para declarar que se usará la variable de carácter `clave` para almacenar un carácter.

4. Escriba instrucciones de declaración para las siguientes variables:
- **a.** `num1`, `num2` y `num3` usados para almacenar números enteros
- **b.** `amps1`, `amps2`, `amps3` y `amps4` usados para almacenar números de precisión doble
- **c.** `voltios1`, `voltios2` y `voltios3` usados para almacenar números de precisión doble
- **d.** `codigoA`, `codigoB`, `codigoC`, `codigoD` y `codigoE` usados para almacenar tipos de carácter

5. Escriba instrucciones de declaración para las siguientes variables:
- **a.** `primernum` y `segundonum` usados para almacenar enteros
- **b.** `velocidad`, `aceleración` y `distancia` usados para almacenar números de precisión doble
- **c.** `empuje` usado para almacenar un número de precisión doble

6. Vuelva a escribir cada una de estas instrucciones de declaración como tres declaraciones individuales.
- **a.** `int mes, día = 30, anio;`
- **b.** `double horas, voltios, potencia = 15.62;`
- **c.** `double precio, cantidad, impuestos;`
- **d.** `char teclaEntrada, ch, opción = 'f';`

7. **a.** Determine el efecto de cada instrucción en el siguiente programa:

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

int main()
{
  int num1, num2, total;
  num1 = 25;
  num2 = 30;
  total = num1 + num2;
  cout << "El total de" << num1 << " y "
       << num2 << " es " << total << endl;

  return 0;
}
```

- **b.** ¿Cuál es la salida que se imprimirá cuando se ejecute el programa mostrado en el ejercicio 7a?

8. Toda variable tiene tres elementos asociados. ¿Cuáles son estos tres elementos?

NOTA PARA LOS EJERCICIOS 9 A 11: Suponga que un carácter requiere un byte de almacenamiento, un entero cuatro bytes, un número de precisión simple cuatro bytes, un número de precisión doble ocho bytes y que a las variables se les asigna almacenamiento en el orden en que son declaradas. (Repase la sección 1.6 si no está familiarizado con el concepto de byte.)

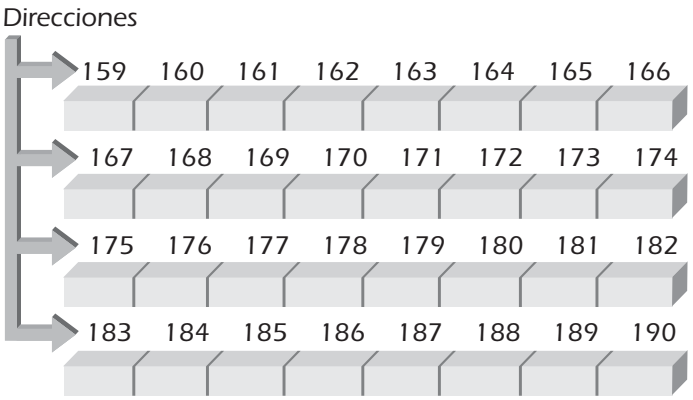


Figura 2.16 *Bytes de memoria para los ejercicios 9, 10 y 11.*

9. a. Usando la figura 2.16 y suponiendo que el nombre de la variable `tasa` se asigna al byte que tiene dirección de memoria 159, determine la dirección correspondiente a cada variable declarada en las siguientes instrucciones. También llene los bytes apropiados con los datos de inicialización incluidos en las instrucciones de declaración. (Use letras para los caracteres, no los códigos de computadora que se almacenarían en realidad.)

```
float tasa;
char ch1 = 'M', ch2 = 'E', ch3 = 'L', ch4 = 'T';
double impuestos;
int num, count = 0;
```

b. Repita el ejercicio 9a, pero sustituya los patrones de byte reales que usaría una computadora que utilice el código ASCII para almacenar los caracteres en las variables `ch1`, `ch2`, `ch3` y `ch4`. (*Sugerencia:* Use el apéndice B.)

10. a. Usando la figura 2.16 y suponiendo que a la variable nombrada `cn1` se le asigna al byte en la dirección de memoria 159, determine las direcciones correspondientes a cada variable declarada en las siguientes instrucciones. Además, llene los bytes apropiados con los datos de inicialización incluidos en las instrucciones de declaración. (Use letras para los caracteres y no los códigos de computadora que se almacenarían en realidad.)

```
char cn1 = 'P', cn2 = 'E', cn3 = 'R', cn4 = 'F', cn5 = 'E';
char cn6 = 'C', cn7 = 'T', key = '\\', sch = '\'', inc = 'A';
char inc1 = 'T';
```

b. Repita el ejercicio 10a, pero sustituya los patrones de byte reales que usaría una computadora que utilice el código ASCII para almacenar los caracteres en cada una de las variables declaradas. (*Sugerencia:* Use la tabla 2.3.)

11. Usando la figura 2.16 y suponiendo que el nombre de variable `millas` se asigna al byte en la dirección de memoria 159, determine las direcciones correspondientes a cada variable declarada en las siguientes instrucciones.

```
float millas;
int cuenta, num;
double dist, temp;
```

2.6 APLICACIÓN DEL PROCEDIMIENTO DE DESARROLLO DE SOFTWARE

Recordará de la sección 1.2 que escribir un programa en C++ es en esencia el tercer paso en el proceso de programación. Los primeros dos pasos en el proceso son determinar lo que se requiere y seleccionar el algoritmo que se va a codificar en C++. En esta sección se muestra cómo los pasos presentados en la sección 1.2 se aplican en la práctica cuando se convierten problemas de programación de trabajo en programas C++. Para repasar, una vez que se establece el requerimiento o problema de un programa, el procedimiento de desarrollo de software consiste de los siguientes pasos:

◆ Paso 1: Analizar el problema

El análisis de un problema puede consistir hasta de dos partes. La primera parte es un ***análisis básico*** que debe realizarse en todos los problemas y consiste en extraer la información completa de entrada y salida proporcionada por los problemas. Es decir, debe:

1. Determinar y entender los elementos de salida deseados que debe producir el programa
2. Determinar los elementos de entrada

Juntos, estos dos elementos se conocen como la entrada/salida del problema, I/O (por sus siglas en inglés), para abreviar. Sólo después que se ha determinado la I/O de un problema es posible seleccionar un algoritmo para transformar las entradas en las salidas deseadas. En este punto, en ocasiones es necesario o útil, o ambas cosas, realizar un cálculo manual para verificar que en efecto puede obtenerse la salida a partir de las entradas. Es claro que si se ha proporcionado una fórmula que relaciona las entradas con las salidas, este paso puede omitirse en esta etapa. Si las entradas requeridas están disponibles y la salida o salidas deseadas pueden producirse, se dice que el problema está definido con claridad y puede resolverse.

Por una variedad de razones puede no ser posible completar un análisis básico. Si esto es así, puede ser necesario un análisis extendido. Un ***análisis extendido*** tan sólo significa que debe obtener información adicional sobre el problema, de modo que pueda entender a fondo lo que se está pidiendo y cómo lograr el resultado. En este texto cualquier información adicional requerida para una comprensión del problema se suministrará junto con el planteamiento de éste.

◆ Paso 2: Desarrollar una solución

Este paso con frecuencia se conoce como paso de diseño, y usaremos los términos ***diseño*** y ***desarrollo*** de forma indistinta. En este paso debe elegir un algoritmo para transformar los elementos de entrada en las salidas deseadas y refinarlo según sea necesario para definir de manera adecuada todas las características que desea que tenga el programa. Si no ha realizado un cálculo manual usando el algoritmo en el paso de análisis, debería hacerlo ahora, usando valores de entrada específicos.

Al diseñar una solución, el enfoque específico que adoptaremos se conoce a menudo como ***enfoque descendente***. Este enfoque consiste en comenzar con la solución más general y refinarla de tal manera que la solución final consista en tareas definidas con claridad que puedan ser completadas por funciones individuales del programa.

◆ Paso 3: Codificar la solución

En este punto se escribe en realidad el programa de C++ que corresponde a la solución desarrollada en el paso 2.

◆ Paso 4: Prueba y corrección del programa

Esto se realiza por medio de datos de prueba seleccionados y se utliza para hacer correcciones al programa cuando se encuentran errores. Un conjunto de datos de prueba que siempre debería usarse son los datos utilizados en su cálculo manual previo.

Para ver cómo pueden aplicarse cada uno de estos pasos en la práctica, ahora los empleamos en el siguiente problema de programación simple.

La resistencia eléctrica, r*, de un alambre metálico, en ohmios, está dada por la fórmula* r = (ml)/a*, donde* m *es la resistividad del metal;* l *es el largo del alambre, en pies; y* a *es el área de corte transversal del alambre, en circular mils. Usando esta información, escriba un programa en C++ para calcular la resistencia de un alambre que mide 125 pies de largo, tiene un área de corte transversal de 500 mils circulares y es de cobre. La resistividad del cobre,* m*, es 10.4.*

Paso 1 Analizar el problema

El primer paso para desarrollar un programa para este problema planteado es realizar un análisis básico. Comenzaremos por determinar las salidas requeridas. Con frecuencia, en el planteamiento del problema usarán palabras como *calcular*, *imprimir*, *determinar*, *encontrar* o *comparar*, las cuales pueden emplearse para determinar las salidas deseadas.

Para el planteamiento del problema de muestra, el enunciado clave es "calcular la resistencia de un alambre". Esto identifica con claridad un elemento de salida. Debido a que no hay otros enunciados así en el problema, sólo se requiere una salida.

Después que se ha identificado con claridad la salida deseada, el paso de análisis básico continúa con la identificación de todos los elementos de entrada. Es esencial en esta etapa distinguir entre elementos de entrada y valores de entrada. Un **elemento de entrada** es el nombre de una cantidad de entrada, mientras un **valor de entrada** es un número o cantidad específica que puede ser el elemento de entrada. Por ejemplo, en el planteamiento del problema de muestra, los elementos de entrada son la resistividad, m, el largo del alambre, l, y el área de corte transversal del alambre, a. Aunque estos elementos de entrada tienen valores numéricos específicos, estos valores de los elementos de entrada por lo general no son de importancia en esta etapa.

La razón por la que los valores de entrada no son necesarios en este punto es que la selección de un algoritmo por lo general es independiente de valores de entrada específicos. El algoritmo depende de saber cuáles son los elementos de salida y entrada y si hay algunos límites especiales. Veamos por qué esto es así.

Del planteamiento del problema es claro que el algoritmo para transformar los elementos de entrada en la salida deseada está dado por la fórmula *r = (ml)/a*. Hay que observar que esta fórmula puede usarse sin tener en cuenta los valores específicos asignados a *m*, *l* o *a*. Aunque no se puede producir un valor numérico real para el elemento de salida (resistencia) a menos que se tengan valores numéricos reales para el elemento de entrada, la relación correcta entre entradas y salidas está expresada por la fórmula. Recuerde que esto es precisamente lo que proporciona un algoritmo: una descripción de cómo las entradas se transforman en salidas que funcione para todas las entradas.

Paso 2 Desarrollar una solución

El algoritmo básico para transformar las entradas en la salida deseada es proporcionado por la fórmula dada. Ahora debe refinarse enumerando, con detalle, cómo se han de combinar las entradas, las salidas y el algoritmo para producir una solución. Este listado indica los pasos que seguirá el programa para resolver el problema. Como tal constituye un esbozo de la forma final que seguirá el código del programa. Usando seudocódigo, el algoritmo completo para resolver este problema es

Asignar valores a m, l y a
Calcular la resistencia usando la fórmula r = (ml)/a
Mostrar la resistencia

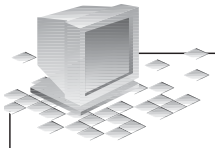
Hay que observar que la estructura de este algoritmo se ajusta a la estructura de control secuencial presentada en la sección 1.2.

Habiendo seleccionado y depurado el algoritmo, el siguiente paso en el diseño (si no se hizo ya en el paso de análisis) es comprobar el algoritmo en forma manual usando datos específicos. Realizar un cálculo manual, ya sea con papel y lápiz o por medio de una calculadora, ayuda a verificar que en realidad ha entendido el problema. Una ventaja adicional de hacer un cálculo manual es que los resultados pueden usarse después para compararlos con la operación del programa en la fase de prueba. Entonces, cuando el programa final se use con otros datos, habrá establecido un grado de confianza en que se está calculando un resultado correcto.

Hacer un cálculo manual requiere que se tengan valores de entrada específicos que puedan ser asignados y usados por el algoritmo para producir la salida deseada. Para este problema se dan tres valores de entrada: una resistividad de 10.4, un área de corte transversal de 500 circular mils y un largo de 125 pies. Al sustituir estos valores en la fórmula, se obtiene una resistencia = (10.4)(125)/500 = 2.60 ohmios para el alambre de cobre.

Paso 3 Codificar la solución

Debido a que se ha desarrollado en forma minuciosa una solución de programa, todo lo que resta es codificar el algoritmo de solución en C++. Esto significa declarar variables de entrada y salida apropiadas, inicializar las variables de entrada en forma correcta, calcular la resistencia e imprimir el valor de resistencia calculado. El programa 2.11 ejecuta estos pasos.



Programa 2.11

```
#include <iostream>
using namespace std;
int main()
{
  double resistividad, area, longitud, resistencia;

  resistividad = 10.4;
  area = 500;
  longitud = 125;
  resistencia = (resistividad * longitud) / area;

  cout << "La resistencia del alambre (en ohmios) es "
          << resistencia << endl;

  return 0;
}
```

Cuando se ejecuta el programa 2.11, se produce la siguiente salida:

```
La resistencia del alambre (en ohmios) es 2.6
```

Ahora que se tiene un programa funcional que produce un resultado, puede comenzar el paso final en el proceso de desarrollo, probar el programa.

Paso 4 Prueba y corrección del programa

El propósito de probar un programa es verificar que funciona en forma correcta y en realidad satisface los requerimientos. Una vez que se ha completado la prueba, el programa puede usarse para calcular salidas para diferentes datos de entrada sin necesidad de volver a probarlo. Éste es, por supuesto, uno de los valores reales de escribir un programa; el mismo programa puede usarse una y otra vez con datos de entrada nuevos.

El método de prueba más simple es verificar la operación del programa con conjuntos de datos de entrada seleccionados cuidadosamente. Un conjunto de datos de entrada que siempre debería usarse son los datos que se seleccionaron para el cálculo manual realizado con anterioridad en el paso 2 del procedimiento de desarrollo. En este caso el programa es relativamente simple y sólo ejecuta un cálculo. Debido a que la salida producida por la ejecución de prueba coincide con el cálculo manual se tiene un buen grado de confianza de que puede ser usado para calcular correctamente la resistencia de otros valores de entrada.

Ejercicios 2.6

NOTA: En cada uno de estos ejercicios se expone un problema de programación. Lea el planteamiento del problema primero y luego responda las preguntas relacionadas con él.

No *escriba un programa para resolver los problemas, sino tan sólo responda las preguntas que siguen a la especificación del programa.*

1. Suponga que tiene que escribir un programa en C++ para calcular la resistencia total de un circuito en serie. En dicho circuito la resistencia total es la suma de todos los valores de resistencia individuales. Suponga que el circuito consiste en una cantidad de resistores de 56 ohmios, 33 ohmios y 15 ohmios.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida. Suponga que la cantidad de resistores de 56 ohmios es *m*, la cantidad de resistores de 33 ohmios es *n*, la cantidad de resistores de 15 ohmios es *p*.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: $m = 17$, $n = 24$ y $p = 12$.

2. Suponga que tiene que escribir un programa para calcular el valor de la *distancia*, en millas, dada la relación:

   $$distancia = velocidad * tiempo\ transcurrido$$

   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: *velocidad* es 55 millas por hora y *tiempo transcurrido* es 2.5 horas.
   e. ¿Cómo debe modificarse el algoritmo que determinó en la parte c si el tiempo transcurrido se diera en minutos en lugar de horas?

3. Suponga que tiene que escribir un programa para determinar el valor de *ergios*, dadas las relaciones:

   $$Ergios = fergios * \sqrt{lergios}$$
   $$Lergios = 2 * \pi * e^{\mu}$$

   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: *fergios* = 14.65, $\pi = 3.1416$, $\mu = 1.672$ y $e = 2.7818$.

4. Suponga que tiene que escribir un programa para mostrar las siguientes especificaciones:

Amplificación de voltaje:	35
Potencia de salida:	2.5 vatios
Ancho de banda:	15KHz

   a. Para este problema de programación, ¿cuántas líneas de salida se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.

5. Escriba un programa en C++ para determinar la distancia recorrida por un automóvil después de 10 segundos, suponiendo que el automóvil viaja inicialmente a 60 millas por hora y el conductor aplica los frenos para desacelerar de manera uniforme a una velocidad de 12 millas/s^2. Use el hecho que *distancia* = $s - (1/2)dt^2$, donde s es la velocidad inicial del automóvil, d es la desaceleración y t es el tiempo transcurrido.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando los datos proporcionados en el problema.

6. Considere el siguiente problema de programación: en 1627, la isla de Manhattan fue vendida a los colonizadores holandeses por aproximadamente 24 dólares. Si las ganancias de esa venta se hubieran depositado en un banco holandés que pagara 5% de interés anual compuesto, ¿cuál sería el saldo principal al final de 2002? Se requiere un despliegue como sigue: El saldo al 31 de diciembre de 2002 es: xxxxxx, donde xxxxxx es la cantidad calculada por su programa.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando los datos proporcionados en el planteamiento del problema.

7. Escriba un programa que calcule y despliegue los voltajes de salida de dos circuitos eléctricos y la suma de los dos voltajes. El voltaje de salida para el primer circuito está dado por la ecuación (150) $V / 0.38f$ y el voltaje de salida para el segundo circuito está dado por la ecuación

$$\frac{230V}{\sqrt{56^2 + (0.98f)^2}}$$

   donde V es el voltaje de entrada al circuito y f es la frecuencia en Hertz.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: el primer circuito es operado con un voltaje de entrada de 1.2 voltios a una frecuencia de 144 Hertz y el segundo circuito es operado con un voltaje de entrada de 2.3 voltios a 100 Hertz.

8. Considere el siguiente problema de programación: 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 calcula y despliega el valor de la desviación normal estándar cuando $X = 85.3$, $\mu = 80$ y $\sigma = 4$.

a. Para este problema de programación, ¿cuántas salidas se requieren?
b. ¿Cuántas entradas tiene este problema?
c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
d. Pruebe el algoritmo escrito para la parte c usando los datos proporcionados en el problema.

9. La ecuación de la curva normal (en forma de campana) usada en aplicaciones estadísticas es

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

Usando esta ecuación, suponga que tiene que escribir un programa en C++ que calcule el valor de y.
a. Para este problema de programación, ¿cuántas salidas se requieren?
b. ¿Cuántas entradas tiene este problema?
c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
d. Pruebe el algoritmo escrito para la parte c suponiendo que $\mu = 90$, $\sigma = 4$, $x = 80$ y $\pi = 3.1416$.

2.7 APLICACIONES

En esta sección, el procedimiento de desarrollo de software presentado en la sección anterior se aplica a dos problemas de programación específicos. Aunque cada problema es diferente, el procedimiento de desarrollo funciona para ambas situaciones. Este procedimiento puede aplicarse a cualquier problema de programación para producir un programa completo y forma el fundamento para todos los programas desarrollados en este texto.

Aplicación 1: Trampas de un radar de velocidad

Un radar común de detección de velocidad de la policía de caminos emite un rayo de microondas a una frecuencia f_0. El rayo es reflejado por un automóvil que se aproxima y el rayo reflejado es captado y analizado por la unidad de radar. La frecuencia del rayo reflejado es cambiada ligeramente de f_0 a f_1 debido al movimiento del automóvil. La relación entre la velocidad del automóvil, v, en millas por hora, y las dos frecuencias de microondas es

$$v = (6.685 \times 10^8)(f_1 - f_0) / (f_1 + f_0)$$

donde las ondas emitidas tienen una frecuencia de $f_0 = 2 \times 10^{10}$ sec^{-1}. Usando la fórmula dada, escriba un programa en C++ para calcular y desplegar la velocidad correspondiente a una frecuencia recibida de 2.000004×10^{10} sec^{-1}.

Ahora aplicamos el procedimiento de desarrollo de software a este problema.

Paso 1 Analizar el problema

Para este problema el programa requiere una sola salida: la velocidad del automóvil. Los elementos de entrada requeridos para solucionar la velocidad son la frecuencia emitida, f_0, y la frecuencia recibida, f_1.

Paso 2 Desarrollar una solución

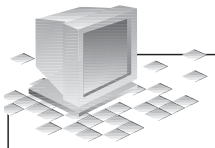
El algoritmo proporcionado para transformar los tres elementos de entrada en el elemento de salida deseado está dado por la fórmula $v = 6.685 \times 10^8(f_1 - f_0)/(f_1 + f_0)$. Por tanto, el algoritmo completo para el programa de solución es

Asignar valores a f_1 y f_0
Calcular la velocidad usando la fórmula $v = 6.685 \times 10^8(f_1 - f_0)/(f_1 + f_0)$
Desplegar la velocidad

Un cálculo manual, usando los datos $f_0 = 2 \times 10^{10}$ sec^{-1} y $f_1 = 2.000004 \times 10^{10}$ sec^{-1} produce una velocidad de 66.85 millas por hora.

Paso 3 Codificar la solución

El programa 2.12 proporciona el código necesario.



Programa 2.12

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

int main()
{
  double velocidad, f0, f1;

  f0 = 2e-10;
  f1 = 2.0000004e-10;

  velocidad = 6.685e8 * (f1 - f0) / (f1 + f0);
  cout << "La velocidad es " << velocidad << "millas/hora " <<endl;

  return 0;
}
```

El programa 2.12 comienza con un comando preprocesador `#include` seguido por una función `main()`. Esta función comienza con la palabra clave `main` y termina con la llave de cierre, }. Además, el programa 2.12 contiene una instrucción de declaración, tres instrucciones de asignación y una instrucción de salida. Las instrucciones de asignación `f0 = 2e-10` y `f1 = 2.0000004e-10` se usan para inicializar las variables `f0` y `f1` respectivamente. La instrucción de asignación

```
velocidad = 6.685e8 * (f1 - f0) / (f1 + f0);
```

calcula un valor para la variable denominada `velocidad`. Cuando el programa 2.12 es compilado y ejecutado se produce la siguiente salida.

```
La velocidad es 66.85 millas/hora
```

Paso 4 Probar y corregir el programa

El último paso en el procedimiento de desarrollo es probar la salida. Debido a que el cálculo único y el valor desplegado concuerdan con el cálculo manual anterior, se ha verificado la operación correcta del programa. Esto permite usar el programa para valores diferentes de frecuencias recibidas. Hay que observar que si los paréntesis no estuvieran colocados en forma correcta en la instrucción de asignación que calculó un valor para la velocidad, el valor desplegado no concordaría con nuestro cálculo manual previo. Esto nos habría alertado del hecho que hubo un error en el programa.

Aplicación 2: Redes de conmutación telefónica

Una red telefónica conectada en forma directa es aquella en la que todos los teléfonos en la red están conectados en forma directa y no requieren una estación de conmutación central para establecer llamadas entre dos de ellos. Por ejemplo, las instituciones financieras en Wall Street usan una red así para mantener líneas telefónicas abiertas en forma directa y continua entre las empresas.

El número de líneas directas necesarias para mantener una red conectada en forma directa para n teléfonos está dado por la fórmula:

$$líneas = n(n - 1)/2$$

Por ejemplo, conectar en forma directa cuatro teléfonos requiere 6 líneas individuales (véase la figura 2.17). Agregar un quinto teléfono a la red ilustrada en la figura 2.17 requeriría 4 líneas adicionales para un total de 10 líneas.

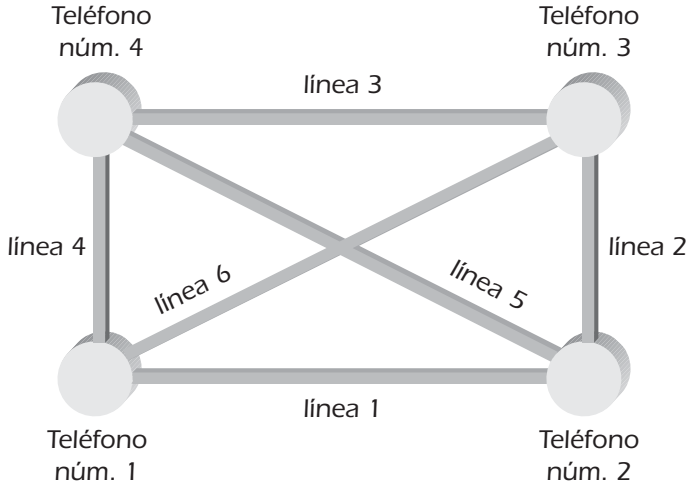


Figura 2.17 Conexión en forma directa de cuatro teléfonos.

Usando la fórmula dada, escriba un programa en C++ que determine el número de líneas directas requeridas para 100 teléfonos, y las líneas adicionales requeridas si se fueran a agregar 10 teléfonos nuevos a la red.

Paso 1 Analizar el problema

Para este programa se requieren dos salidas: el número de líneas directas para 100 teléfonos y el número de líneas adicionales necesarias cuando se agregan 10 teléfonos nuevos a la red existente. El elemento de entrada requerido para este problema es el número de teléfonos, el cual se denota como *n* en la fórmula.

Paso 2 Desarrollar una solución

La primera salida se obtiene con facilidad usando la fórmula:

$líneas = n(n - 1)/2$

Aunque no se proporciona una fórmula para las líneas adicionales, puede usarse la fórmula dada para determinar el número total de líneas necesarias para 110 suscriptores. Restar el número de líneas para 100 suscriptores del número de líneas necesarias para 110 suscriptores producirá entonces el número de líneas adicionales requeridas. Por tanto, el algoritmo completo para el programa, en seudocódigo, es

Calcular el número de líneas directas para 100 suscriptores
Calcular el número de líneas directas para 110 suscriptores
Calcular las líneas adicionales necesarias, que son la diferencia entre el segundo y el primer cálculo.
Desplegar el número de líneas para 100 suscriptores
Desplegar las líneas adicionales necesarias

Verificar este algoritmo en forma manual, usando los datos proporcionados, produce la respuesta:

$líneas = 100(100 - 1)/2 = 100(99)/2 = 4950$

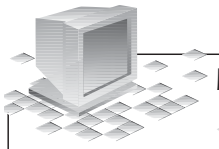
para 100 teléfonos y

$líneas = 5995$

para 110 teléfonos. Por tanto, serían necesarias 1045 líneas adicionales para conectar en forma directa los diez teléfonos adicionales en la red existente.

Paso 3 Codificar la solución

El programa 2.13 proporciona el código necesario.



Programa 2.13

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

int main()
{
  int numin1, numin2, lineas1, lineas2;

  numin1 = 100;
  numin2 = 110;
  lineas1 = numin1 * (numin1 - 1)/2;
  lineas2 = numin2 * (numin2 - 1)/2;
  cout << "El número de líneas iniciales es " << lineas1 << "." << endl;
  cout << "Se necesitan " << lineas2 - lineas1
       << " líneas adicionales." << endl;

  return 0;
}
```

Como antes, el programa en C++ incluye el archivo de encabezado `iostream` y consiste en una función `main()`. El cuerpo de esta función comienza con la llave de apertura {, y termina con la llave de cierre}. Debido a que el número de líneas entre suscriptores debe ser un número entero (no es posible una línea fraccionaria) las variables `lineas1` y `lineas2` se especifican como variables enteras. Las primeras dos instrucciones de asignación inicializan las variables `numin1` y `numin2`. La siguiente instrucción de asignación calcula el número de líneas necesarias para 100 suscriptores y la última instrucción de asignación calcula el número de líneas para 110 suscriptores. La primera instrucción `cout` se usa para desplegar un mensaje y el resultado del primer cálculo. La siguiente instrucción `cout` se usa para desplegar la diferencia entre los dos cálculos. La siguiente salida se produce cuando se compila y ejecuta el programa 2.13:

```
El número de líneas iniciales es 4950.
Se necesitan 1045 líneas adicionales.
```

Paso 4 Probar y corregir el programa

Como los dos cálculos y los valores desplegados concuerdan con el cálculo manual previo, se ha verificado la operación correcta del programa.

Ejercicios 2.7

1. **a.** Modifique el programa 2.12 para calcular la velocidad de un automóvil cuya frecuencia de retorno del radar es $2.00000035 \times 10^{10}$ sec^{-1}.
   b. Compile y ejecute el programa escrito para el ejercicio 1a en una computadora.

2. a. Modifique el programa 2.12 para determinar la frecuencia que será regresada por un automóvil que viaja a 55 millas por hora. Su programa deberá producir el siguiente despliegue:

```
La frecuencia regresada correspondiente a 55 millas por hora
es_____
```

donde el subrayado es reemplazado por el valor real calculado por su programa.

b. Compile y ejecute el programa escrito para el ejercicio 2a en una computadora. Asegúrese de hacer un cálculo manual de modo que pueda verificar los resultados producidos por su programa.

c. Después de haber verificado los resultados del programa escrito en ejercicio 2a, modifique el programa para calcular la frecuencia de vuelta para un automóvil que viaja a 75 millas por hora.

3. a. Modifique el programa 2.13 para calcular y desplegar el número total de líneas necesario para conectar 1000 teléfonos individuales en forma directa entre sí.

b. Compile y ejecute el programa escrito para el ejercicio 3a en una computadora.

4. a. Modifique el programa 2.13 de modo que una nueva variable `numfin`, el número adicional de suscriptores que va a ser conectados a la red existente, se inicialice en 10. Haga otros cambios en el programa de modo que produzca el mismo despliegue que el programa 2.13.

b. Compile y ejecute el programa escrito para el ejercicio 4a en una computadora. Verifique que el despliegue producido por su programa corresponde al despliegue mostrado en el texto.

5. a. Diseñe, escriba, compile y ejecute un programa en C++ para convertir temperatura en grados Fahrenheit en grados Celsius. La ecuación para esta conversión es

Celsius = 5.0/9.0 (*Fahrenheit* - 32.0).

Haga que su programa convierta y despliegue la temperatura Celsius correspondiente a 98.6 grados Fahrenheit. Su programa deberá producir el siguiente despliegue:

```
Para una temperatura Fahrenheit de_______ grados,
  la temperatura Celsius equivalente es_______ grados.
```

donde su programa debe insertar los valores apropiados en lugar de los subrayados.

b. Verifique en forma manual los valores calculados por su programa. Después que haya comprobado que su programa funciona en forma correcta, modifíquelo para convertir 86.5 grados Fahrenheit en su valor Celsius equivalente.

6. a. Escriba, compile y ejecute un programa en C++ para calcular la resistencia de un circuito en serie consistente en 12 resistores de 56 ohmios, veinte de 39 ohmios, 32 de 27 ohmios y 27 de 15 ohmios. Use el hecho que la resistencia de un circuito en serie es la suma de todas las resistencias individuales. Su programa deberá producir el siguiente despliegue:

```
La resistencia total, en ohmios, es xxxx
```

donde xxxx es reemplazado por el valor de resistencia real calculado por su programa.

b. Verifique en forma manual los valores calculados por su programa. Después que haya comprobado que su programa funciona en forma correcta, modifíquelo para calcular la resistencia de un circuito en serie consistente en ningún resistor de 56 ohmios, 17 de 39 ohmios, 19 de 27 ohmios y 42 de 15 ohmios.

7. a. Diseñe, escriba, compile y ejecute un programa en C++ para calcular el tiempo requerido para hacer un viaje de 183.67 millas. La ecuación para calcular el tiempo transcurrido es

tiempo transcurrido = distancia total / velocidad promedio

Suponga que la velocidad promedio durante el viaje fue 58 millas por hora.

b. Verifique en forma manual los valores calculados por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar el tiempo que toma hacer un viaje de 372 millas a una velocidad promedio de 67 millas por hora.

8. a. Diseñe, escriba, compile y ejecute un programa en C++ para calcular la suma de los números del 1 al 100. La fórmula para calcular esta suma es

$$suma = (n/2)\ (\ 2*a + (n - 1)d)$$

donde n = número de términos que se van a sumar, a = el primer número y d = la diferencia entre cada número.

b. Verifique en forma manual los valores calculados por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la suma de los enteros de 100 a 1000.

NOTA: Los ejercicios 9 a 13 requieren elevar un número a una potencia. Esto puede lograrse usando la función de potencia de C++ `pow()`. *Por ejemplo, la instrucción* `pow(2.0,5.0);` *eleva el número 2.0 a la quinta potencia, y la instrucción* `pow(num1,num2);` *eleva la variable* `num1` *a la potencia* `num2`. *Para usar la función de potencia se coloca un comando preprocesador* `#include <cmath>` *en una sola línea después del comando* `#include <iostream>` *o se incluye la instrucción de declaración* `double pow();` *con las instrucciones de declaración de variables usadas en su programa. La función de potencia se explicará con más detalle en la sección 3.3.*

9. a. La ley de Newton del enfriamiento establece que cuando un objeto con una temperatura inicial T se introduce en una sustancia de temperatura A, alcanzará una temperatura $TFIN$ en t minutos de acuerdo con la

$$TFIN = (T - A)\ e^{-kt} + A$$

En esta fórmula e es el número irracional 2.71828 redondeado a cinco lugares decimales, conocido por lo común como el número de Euler, y k es un coeficiente térmico, el cual depende del material que se va a enfriar. Usando esta fórmula escriba, compile y ejecute un programa en C++ que determine la temperatura alcanzada por un objeto después de 20 minutos cuando es colocado en un vaso con agua cuya temperatura es de 60 grados. Suponga que el objeto tenía inicialmente una temperatura de 150 grados y tiene una constante térmica de 0.0367.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la temperatura alcanzada después de 10 minutos cuando es colocado en un vaso con agua cuya temperatura es de 50 grados.

10. a. La ganancia de voltaje de un amplificador está dada por la fórmula

$$\textit{ganancia de voltaje} = [275 / (23^2 + (0.5f)^2)^{1/2}]^n$$

donde f es la frecuencia, en Hertz, y n es el número de etapas en el amplificador. Usando esta fórmula escriba, compile y ejecute un programa en C++ que determine el valor de la ganancia de voltaje para un amplificador de cuatro etapas que opera a una frecuencia de 120 Hertz. Su programa deberá producir el siguiente despliegue:

```
A una frecuencia de xxxxx hertz, la ganancia de voltaje
es yyyyy
```

donde xxxxx es reemplazado por la frecuencia e yyyyy por la ganancia de voltaje.

b. Verifique en forma manual el valor calculado por su programa. Después que haya comprobado que su programa funciona en forma correcta, modifíquelo para determinar la ganancia de voltaje de un amplificador de 12 etapas que opera a una frecuencia de 9500 Hertz.

11. a. La corriente eléctrica, i, en amperios, que fluye a través del circuito ilustrado en la figura 2.18 está dada por la

$$i = \frac{E(1-e)^{-(R/L)t}}{R}$$

donde E es el voltaje de la batería en voltios, R es el valor del resistor en ohmios, L es el valor del inductor en henrios, t es el tiempo en segundos después de cerrar el interruptor y e es el número de Euler, el cual es 2.718 con una precisión de tres cifras decimales. Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar el flujo de corriente en el circuito ilustrado en la figura 2.18 cuando t es 0.12 segundos.

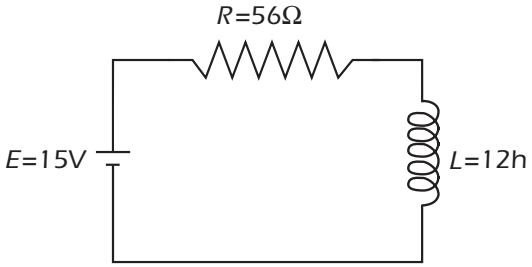


Figura 2.18 Un circuito RL en serie.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la corriente en 0.12 segundos si E es de 25 voltios, R es de 33 ohmios y L es de 15 henrios.

12. a. La corriente eléctrica, i, en amperios, que fluye a través del circuito ilustrado en la figura 2.19 está dado por la siguiente ecuación:

$$i = \frac{(E)e^{-t/RC}}{R}$$

donde E es el voltaje de la batería en voltios, R es el valor del resistor en ohmios, C es el valor del capacitor en faradios, t es el tiempo en segundos después de cerrar el interruptor y e es el número de Euler, el cual es 2.718 con una precisión de tres cifras decimales. Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar el voltaje a través del capacitor ilustrado en la figura 2.19 cuando t es 0.31 segundos.

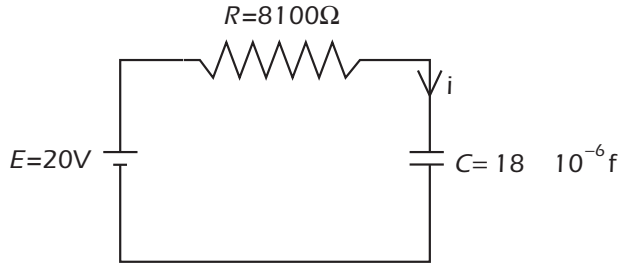


Figura 2.19 Un circuito RC en serie.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la corriente en 0.85 segundos si E es de 25 voltios, R es de 220 ohmios y C es de 0.00039 faradios.

13. a. El voltaje eléctrico, V, en voltios, a través del capacitor, C, ilustrado en la figura 2.20 está dado por la ecuación

$$V = \frac{E\left[1 - e^{-t/RC}\right]}{R}$$

donde E es el voltaje de la batería en voltios, R es el valor del resistor en ohmios, C es el valor del capacitor en faradios, t es el tiempo en segundos después de cerrar el interruptor y e es el número de Euler, el cual es 2.718 con una precisión de tres cifras decimales. Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar el voltaje a través del capacitor ilustrado en la figura 2.20 cuando t es 0.42 segundos.

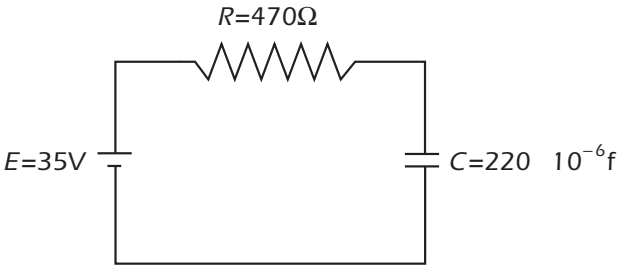


Figura 2.20 Un circuito RC en serie.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la corriente en 0.85 segundos si E es de 25 voltios, R es de 220 ohmios y C es de 0.00039 faradios.

14. a. El conjunto de ecuaciones lineales

$$a_{11}X_1 + a_{12}X_2 = c_1$$
$$a_{21}X_1 + a_{22}X_2 = c_2$$

puede resolverse usando la regla de Cramer:

$$X_1 = \frac{c_1 a_{22} - c_2 a_{12}}{a_{11} a_{22} - a_{12} a_{21}}$$
$$X_2 = \frac{c_2 a_{11} - c_1 a_{21}}{a_{11} a_{22} - a_{12} a_{21}}$$

Usando estas ecuaciones, escriba, compile y ejecute un programa en C++ para encontrar los valores X_1 y X_2 que satisfagan las siguientes ecuaciones:

$$3X_1 + 4X_2 = 40$$
$$5X_1 + 2X_2 = 34$$

b. Verifique en forma manual los valores calculados por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para resolver el siguiente conjunto de ecuaciones:

$$3X_1 + 12.5X_2 = 22.5$$
$$4.2X_1 - 6.3X_2 = 30$$

2.8 ERRORES COMUNES DE PROGRAMACIÓN

Parte de aprender cualquier lenguaje de programación es cometer los errores elementales que se encuentran por lo común cuando se empieza a usar el lenguaje. Estos errores tienden a ser bastante frustrantes debido a que cada lenguaje tiene su propio conjunto de errores de programación comunes esperando a los incautos. Los errores más comunes cometidos cuando se empieza a programar en C++ incluyen los siguientes.

1. Omitir los paréntesis después de `main`.
2. Omitir o escribir de manera incorrecta la llave de apertura `{` que indica el inicio de un cuerpo de función.
3. Omitir o escribir de manera incorrecta la llave de cierre `}` que indica el final de una función.
4. Escribir mal el nombre de un objeto o función; por ejemplo, escribir `cot` en lugar de `cout`.
5. Olvidar cerrar una cadena enviada a `cout` con un símbolo de comillas.
6. Olvidar separar flujos de datos individuales pasados a `cout` con un símbolo de inserción ("enviar a"), `<<`.
7. Omitir el punto y coma al final de cada instrucción de C++.

8. Agregar un punto y coma al final del comando preprocesador `#include`.
9. Olvidar `\n` para indicar una línea nueva.
10. Escribir en forma incorrecta la letra O en lugar del número cero (0), o viceversa. Escribir de forma incorrecta la letra l, por el número 1, o viceversa.
11. Olvidar declarar todas las variables usadas en el programa. Este error es detectado por el compilador y se genera un mensaje de error para todas las variables no declaradas.
12. Almacenar un tipo de datos inapropiado en una variable declarada. Este error es detectado por el compilador y el valor asignado es convertido al tipo de datos de la variable a la que fue asignado.
13. Usar una variable en una expresión antes que se haya asignado un valor a la variable. Aquí, cualquier valor que resulte estar en la variable se usará cuando se evalúa la expresión, y el resultado carecerá de significado.
14. Dividir valores enteros en forma incorrecta. Este error se disfraza por lo general dentro de una expresión más grande y puede ser un error muy problemático de detectar. Por ejemplo, la expresión

    3.425 + 2/3 + 7.9

    produce el mismo resultado que la expresión

    3.425 + 7.9

    debido a que la división de números enteros de 2/3 es 0.
15. Mezclar tipos de datos en la misma expresión sin entender con claridad el efecto producido. En vista que C++ permite expresiones con tipos de datos "mixtos", es importante entender el orden de evaluación y el tipo de datos de todos los cálculos intermedios. Como una regla general, es mejor no mezclar nunca tipos de datos en una expresión a menos que se desee un efecto específico.

    El tercero, quinto, séptimo, octavo y noveno errores en esta lista son los más comunes al inicio, mientras incluso programadores experimentados en ocasiones cometen el décimo error. Vale la pena que escriba un programa e introduzca de manera específica cada uno de estos errores, uno a la vez, para ver qué mensajes de error son producidos por el compilador, si es que se genera alguno. Entonces cuando aparezcan estos mensajes de error debido a errores inadvertidos, tendrá experiencia para comprender los mensajes y corregir los errores.

    En un nivel más fundamental, un error de programación importante cometido por los programadores principiantes es la prisa por codificar y ejecutar un programa antes que el programador haya entendido por completo lo que se requiere y los algoritmos y procedimientos que se usarán para producir el resultado deseado. Un síntoma de este apresuramiento por introducir un programa en la computadora es la carencia de un esbozo del programa propuesto o de un programa escrito en sí. Pueden evitarse muchos problemas con sólo revisar una copia del programa, ya sea manuscrita o enlistada por la computadora, antes que sea compilado.

2.9 RESUMEN DEL CAPÍTULO

1. Un programa en C++ consiste de uno o más módulos llamados funciones. Una de estas funciones debe llamarse `main()`. La función `main()` identifica el punto de inicio de un programa C++.
2. El programa C++ más simple consiste en una sola función `main()`.
3. Después del nombre de la función, el cuerpo de una función tiene la siguiente forma general:

```
{
   Todas las instrucciones de C++ van aquí;
}
```

4. Todas las instrucciones de C++ deben terminar con un punto y coma.
5. En este capítulo se introdujeron tres tipos de datos: enteros, de punto flotante y booleanos. C++ reconoce cada uno de estos tipos de datos, además de otros que aún falta por presentar.
6. El objeto `cout` puede usarse para desplegar todos los tipos de datos de C++.
7. Cuando se usa el objeto `cout` dentro de un programa, debe colocarse el comando preprocesador `#include <iostream>` en la parte superior del programa. Los comandos preprocesadores no terminan con punto y coma.
8. Toda variable en un programa C++ debe declararse como el tipo de valor que puede almacenar. Las declaraciones dentro de una función pueden colocarse en cualquier parte dentro de la función, aunque una variable sólo puede ser usada después de ser declarada. Las variables también pueden inicializarse cuando son declaradas. Además, las variables del mismo tipo pueden declararse usando una sola instrucción de declaración. Las instrucciones de declaración de variables tienen la forma general:

   tipo-de-datos nombre(s)DeVariables;

9. Un programa C++ simple que contiene instrucciones de declaración tiene la forma típica

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

int main()
{
  instrucciones de declaración;

  otras instrucciones;

  return 0;
}
```

Aunque las instrucciones de declaración pueden colocarse en cualquier parte dentro del cuerpo de la función, sólo puede utilizarse una variable después que se ha declarado.

10. Las instrucciones de declaración siempre efectúan una tarea de software que consiste en informar al compilador los nombres válidos de las variables de una función. Cuando una declaración de variable también causa que la computadora asigne ubicaciones de memoria para la variable, la instrucción de declaración también se llama instrucción de definición. (Todas las declaraciones que se han usado en este capítulo han sido también instrucciones de definición.)
11. El operador `sizeof()` puede utilizarse para determinar la cantidad de almacenamiento reservado para las variables.

Consideración de opciones de carrera

Ingeniería eléctrica

La ingeniería eléctrica tiene que ver con la aplicación de los principios de la electricidad y el electromagnetismo para la fabricación de toda clase de máquinas y dispositivos que utilicen electricidad o produzcan energía eléctrica. Este campo es el más grande de todos los campos de la ingeniería. En sus inicios, a mediados del siglo XIX, sólo se interesaba por la generación de energía eléctrica. Ha evolucionado para convertirse en un campo con horizontes más amplios, abarcando las comunicaciones, las computadoras, la robótica, los dispositivos de estado sólido y el diseño de circuitos integrados.

1. Potencia. Esta área implica la generación de energía eléctrica en grandes plantas de combustibles fósiles, nucleares, solares o hidroeléctricas, o la utilización eficiente de la energía eléctrica por medio de motores o dispositivos de iluminación. También son importantes la transmisión y distribución de energía eléctrica por medio de líneas eléctricas aéreas, microondas, ductos y líneas de superconductores.
2. Electrónica de estado sólido. A través de la física moderna y la ciencia de los materiales, se han desarrollado materiales semiconductores exóticos que se usan para construir microcircuitos para la vigilancia y control de las operaciones de toda clase de dispositivos, desde juegos de video hasta robots en líneas de montaje. La mejora en la confiabilidad, la rápida reducción en el tamaño y la reducción en los requerimientos de potencia de los componentes eléctricos miniaturizados modernos han creado oportunidades ilimitadas para sus aplicaciones.
3. Comunicaciones. Las comunicaciones implican el diseño y construcción de equipo usado en la transmisión de información por medio de electricidad u ondas electromagnéticas (radio, luz, microondas, etc.). El uso del láser para la comunicación es un tema de interés contemporáneo, mientras que las características de las antenas y el radar son un poco más antiguas.
4. Computadoras y robótica. Aunque la electrónica tiene que ver con los principios asociados con las funciones de los componentes, los ingenieros en computación están interesados en diseñar los circuitos complejos que entrelazan los componentes en una computadora. Los microprocesadores, o computadoras pequeñas, están diseñados para vigilar y controlar de manera constante las operaciones de una pieza de equipo particular como un torno o un piloto automático.