CAPÍTULO 7
Completar lo básico

TEMAS

El estándar ANSI/ISO actual para C++ introduce dos características nuevas que no eran parte de la especificación original de C++: el manejo de excepciones y una clase `string`. *Ambas características se presentan en este capítulo.*

El manejo de excepciones es un medio de detección y procesamiento de errores, el cual ha obtenido una aceptación creciente en la tecnología de programación. Permite detectar un error en el punto en el código en que ha ocurrido y proporciona un medio de procesarlo y regresar el control a la línea que lo generó. Aunque dicha detección de errores y corrección del código es posible usando instrucciones y funciones `if`, *el manejo de excepciones proporciona una herramienta de programación más útil dirigida a la detección y procesamiento de errores.*

Con el nuevo estándar ANSI/ISO de C++, una clase llamada `string` *es parte ahora de la biblioteca C++ estándar. Esta clase proporciona un conjunto muy amplio*

de funciones de clase que incluye la inserción y eliminación fácil de caracteres de una cadena, la expansión automática de la cadena siempre que se exceda su capacidad original, la contracción de la cadena cuando se eliminen caracteres, y comprobación del rango para detectar posiciones de caracteres inválidas.

Además de presentar estas dos características nuevas de C++, este capítulo muestra cómo el manejo de excepciones, cuando se aplica a las cadenas, proporciona un medio muy útil para validar la entrada del usuario.

7.1 MANEJO DE EXCEPCIONES

El enfoque tradicional de C++ para el manejo de errores usa una función para devolver un valor específico para indicar operaciones específicas. Por lo general, se usa un valor devuelto de 0 o 1 para indicar una ejecución exitosa de la tarea de la función, mientras que se usa un valor negativo para indicar una condición de error. Por ejemplo, si se utiliza una función para dividir dos números, podría usarse un valor devuelto de −1 para indicar que el denominador era cero y que la división no podía llevarse a cabo. Cuando pueden ocurrir múltiples condiciones de error, se usarán diferentes valores de devolución para indicar errores específicos.

Aunque este enfoque aún está disponible y se usa con frecuencia, pueden ocurrir varios problemas con este método. Primero, el programador debe verificar el valor de devolución para detectar si ocurrió un error. A continuación, el código para el manejo de error que verifica el valor devuelto con frecuencia se entremezcla con el código de procesamiento normal, dificultando a veces determinar cuál parte del código está manejando errores y el procesamiento normal del programa. Por último, devolver una condición de error de una función significa que la condición debe ser del mismo tipo de datos que un valor devuelto válido; por tanto, el código de error debe ser un valor identificado de manera especial que pueda ser reconocido como una alerta de error. Esto significa que el código de error está incrustado de manera efectiva como uno de los valores de no error posibles que pueden ser requeridas desde la función y sólo está disponible en el punto donde la función devuelve un valor. Una función que devuelve un valor booleano no tiene valores adicionales que puedan utilzarse para reportar una condición de error.

Nada de esto es insuperable y muchas veces este enfoque es simple y efectivo. Sin embargo, en las versiones más recientes, los compiladores de C++ han agregado una técnica diseñada de manera específica para la detección y manejo de errores conocida como manejo de excepciones.

En el ***manejo de excepciones,*** cuando ocurre un error mientras se está ejecutando una función, el método crea un valor, variable u objeto, el cual se conoce como ***excepción,*** que contiene información sobre el error en el punto en que ocurre. Esta excepción es transmitida de inmediato, en el punto en que fue generada, al código que se denomina ***manejador de excepciones,*** el cual está diseñado para ocuparse de la excepción. El proceso de generar y transmitir la excepción en el punto en que se detectó el error se conoce como ***lanzar una excepción.*** La excepción es lanzada desde dentro de la función mientras aún se está ejecutando. Esto permite manejar el error y luego regresar el control de nuevo a la función de modo que pueda completar su tarea.

En general, dos tipos de errores fundamentales pueden causar excepciones en C++: aquellos que resultan de una incapacidad del programa para obtener un recurso requerido, y aquellos que resultan de datos defectuosos. Son ejemplos del primer tipo de error los

intentos de obtener un recurso del sistema, como localizar y encontrar un archivo para entrada. Estos tipos de errores son el resultado de recursos externos sobre los cuales el programador no tiene control.

El segundo tipo de error puede ocurrir cuando un programa le indica al usuario que introduzca un número entero, y el usuario introduce una cadena, como e234, que no puede ser convertida en un valor numérico. Otro ejemplo es el intento de dividir dos números cuando el denominador tiene un valor de 0. Esta última condición se conoce como error por división entre cero. Cada uno de estos errores siempre puede verificarse y manejarse de una manera que no produzca una falla del programa. Antes de ver cómo se logra esto usando el manejo de excepciones, revise la tabla 7.1 para familiarizarse con la terminología usada en relación con las excepciones de procesamiento.

Tabla 7.1 Terminología del manejo de excepciones

Terminología	Descripción
Excepción	Un valor, variable u objeto que identifica un error específico que ha ocurrido mientras se está ejecutando un programa
Lanzar una excepción	Envía la excepción a una sección de código que procesa el error detectado
Atrapar o manejar una excepción	Recibe una excepción lanzada y la procesa
Cláusula catch	La sección de código que procesa el error
Manejador de excepción	El código que lanza y atrapa una excepción

La sintaxis general del código requerido para lanzar y atrapar una excepción es la siguiente:

```
try
{
  // una o mas instrucciones,
  // al menos una de las cuales debería
  // ser capaz de lanzar una excepción;
}
catch(Tipo-de-datos-de-excepción nombre-de-parámetro)
{
  // una o más instrucciones
}
```

Este ejemplo usa dos nuevas palabras clave: `try` y `catch`.

La palabra clave `try` identifica el inicio de un bloque de código para manejar una excepción. Al menos una de las instrucciones dentro de las llaves que definen a este bloque de código deberá ser capaz de lanzar una excepción. Como ejemplo, examínese el bloque `try` en la siguiente sección de código:

```
try
{
  cout << "Introduzca el numerador (sólo números enteros): ";
  cin >> numerador;
  cout << "Introduzca el denominador (sólo números enteros): ";
  cin >> denominador;
  resultado = numerador/denominador;
}
```

El bloque `try` contiene cinco instrucciones, tres de las cuales pueden producir un error que usted desea descubrir. En particular, un programa escrito de manera profesional dbería asegurar que se introducen números enteros válidos en respuesta a ambos indicadores y que el segundo valor introducido no sea un cero. Para este ejemplo, sólo comprobará que el segundo valor introducido no es un cero. (En el apéndice C encontrará el código de manejo de excepciones que puede usarse para validar ambas entradas y asegurar que ambos datos introducidos son números enteros.)

Desde el punto de vista del bloque `try`, sólo importa el valor del segundo número. El bloque `try` se alterará para expresar "pruebe todas las instrucciones para ver si ocurre una excepción, la cual en este caso particular es un segundo valor cero". Para comprobar que el segundo valor no es un cero, se agrega una instrucción `throw` dentro del bloque `try`, como sigue:

```
try
{
  cout << "Introduzca el numerador (sólo números enteros): ";
  cin >> numerador;
  cout << "Introduzca el denominador (sólo números enteros): ";
  cin >> denominador;
  if (denominador == 0)
    throw denominador;
  else
    resultado = numerador/denominador;
}
```

En este bloque `try`, el elemento lanzado es un valor entero. Podría haberse usado un literal de cadena, una variable o un objeto, pero sólo uno de estos elementos puede ser lanzado por cualquier instrucción throw individual. Las primeras cuatro instrucciones en el bloque `try` no tienen que incluirse en el código; sin embargo, hacerlo así mantiene juntas todas las instrucciones relevantes. Mantener juntas las instrucciones relacionadas puede facilitar agregar instrucciones de lanzamiento dentro del mismo bloque `try` para asegurar que ambos valores de entrada son valores enteros, así que es más conveniente tener todo el código relevante disponible dentro del mismo bloque `try`.

Un bloque `try` debe ir seguido por uno o más bloques `catch`, los cuales sirven como manejadores de excepciones para cualesquiera excepciones lanzadas por las instrucciones en el bloque `try`. Aquí hay un bloque `catch` que maneja la excepción lanzada, la cual es un número entero:

```
catch(int e)
  {
    cout << "Un valor del denominador de " << e << " es inválido." << endl;
    exit (1);
  }
```

El manejo de excepciones proporcionado por este bloque `catch` es una instrucción de salida que identifica la excepción particular descubierta y luego termina la ejecución del programa. Observe los paréntesis que siguen a la palabra clave `catch`. Dentro de los paréntesis se enlista el tipo de datos de la excepción que es lanzada y un nombre de parámetro (el cual es `e`) que se usa para recibirlo. Este identificador, el cual es seleccionado por el programador pero que usa por convención la letra `e` por excepción, se usa para mantener el valor de la excepción generado cuando se lanza una excepción.

Pueden proporcionarse múltiples bloques `catch` en tanto cada bloque atrape un tipo de datos único. El requisito es que al menos se proporcione un bloque `catch` para cada bloque `try`. Entre más excepciones puedan ser descubiertas con el mismo bloque `try`, es mejor. El programa 7.1 proporciona un programa completo que incluye un bloque `try` y un bloque `catch` para detectar un error de división entre cero.

Programa 7.1

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

int main()
{
      int numerador, denominador;

  try
  {
    cout << "Introduzca el numerador (sólo números enteros): ";
    cin >> numerador;
    cout << "Introduzca el denominador (sólo números enteros): ";
    cin >> denominador;
      if (denominador == 0)
        throw denominador; // se lanza un valor entero
    else
        cout << numerador << '/' << denominador
             << " = " << double(numerador)/double(denominador) << endl;
  }
  catch(int e)
  {
    cout << "Un valor del denominador de " << e << " es inválido." << endl;
    exit (1);
  }

  return 0;
}
```

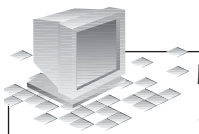
A continuación se encuentran dos muestras de ejecución usando el programa 7.1. Hay que observar que la segunda salida indica que se ha detectado con éxito un intento de dividir entre un denominador de cero antes que se realice la operación.

```
Introduzca el numerador (sólo números enteros): 12
Introduzca el denominador (sólo números enteros): 3
12/3 = 4
```

e

```
Introduzca el numerador (sólo números enteros): 12
Introduzca el denominador (sólo números enteros): 0
Un valor del denominador de 0 es inválido.
```

Habiendo detectado un denominador de cero, en lugar de terminar la ejecución del programa, un programa más robusto puede proporcionarle al usuario la oportunidad de reintroducir un valor diferente de cero. Esto puede lograrse incluyendo el bloque `try` dentro de una instrucción `while` y luego hacer que el bloque `catch` devuelva el control del programa a la instrucción `while` después de informar al usuario que se ha introducido un valor de cero. El siguiente código logra esto:



Programa 7.2

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

int main()
{
  int numerador, denominador;
  bool Denominador_necesario = verdadero;

  cout << "Introduzca el numerador (sólo números enteros): ";
  cin >> numerador;

  cout << "Introduzca el denominador (sólo números enteros): ";
  while(Denominador_necesario)
  {
    cin >> denominador;
    try
    {
      if (denominador == 0)
        throw denominador;  // se lanza un valor entero
    }
    catch(int e)
    {
      cout << "Un valor del denominador de " << e << " es inválido." << endl;
      cout << "Por favor reintroduzca el denominador (solo números enteros): ";
      continue;  // esto regresa el control a la instruccion while
    }
    cout << numerador << '/' << denominador
         << " = " << double(numerador)/double(denominador) << endl;
      Denominador_necesario = falso;
  }

  return 0;
}
```

Al revisar este código, hay que observar que es la instrucción `continue` dentro del bloque `catch` la que devuelve el control a la parte superior de la instrucción `while` (véase la sección 6.3 para una revisión de la instrucción `continue`). A continuación hay una muestra de ejecución usando el programa 7.2:

```
Introduzca el numerador (sólo números enteros): 12
Introduzca el denominador (sólo números enteros): 0
Un valor del denominador de 0 es inválido.
Por favor reintroduzca el denominador (sólo números enteros): 5
12/5 = 2.4
```

Debe hacerse una advertencia cuando se lancen literales de cadena en lugar de valores numéricos. Siempre que se lance un literal de cadena, se trata de una cadena C, no un objeto de clase `string`. Esto significa que la instrucción `catch` debe declarar el argumento recibido como una cadena C, la cual es un arreglo de caracteres, en vez de una cadena. Como ejemplo, considérese que en lugar de lanzar el valor de la variable `denominador` en los programas 7.1 y 7.2, se usa la siguiente instrucción:

```
throw "***Entrada invalida – No se permite un valor del denominador de cero***";
```

Aquí se encuentra una instrucción `catch` correcta para la instrucción `throw` precedente:

```
catch(char e[])
```

Un intento de declarar la excepción como una variable de clase `string` producirá un error de compilador.

Ejercicios 7.1

1. Defina los siguientes términos:

   excepción
   bloque `try`
   bloque `catch`
   manejador de excepción
   lanzar una excepción
   atrapar una excepción

2. Introduzca y ejecute el programa 7.1.

3. Reemplace la instrucción

   ```
   cout << numerador << '/' << denominador
        << " = " << double(numerador)/double(denominador) << endl;
   ```

   en el programa 7.1 con la instrucción

   ```
   cout << numerador << '/' << denominador
        << " = " << numerador/denominador << endl;
   ```

   y ejecute el programa modificado. Introduzca los valores 12 y 5 y explique por qué es incorrecto el resultado desde el punto de vista del usuario.

4. Modifique el programa 7.1 de modo que lance y atrape el mensaje `***Entrada invalida – No se permite un valor del denominador de cero***.` (*Sugerencia:* Revise la advertencia presentada al final de esta sección.)
5. Introduzca y ejecute el programa 7.2.
6. Modifique el programa 7.2 de modo que continúe dividiendo dos números hasta que el usuario introduzca el carácter q (como numerador o denominador) para terminar la ejecución del programa.
7. Incluya el código de manejo de excepciones proporcionado en esta sección dentro del programa 7.1 para asegurar que el usuario introduce un valor entero válido tanto para el numerador como para el denominador.

7.2 LA CLASE `string`

Los programas en este texto han usado el objeto `cout` de la clase `istream` en forma extensa sin haber investigado esta clase o la manera en que se crea el objeto `cout`. Ésta es una de las ventajas del diseño de programas orientado a objetos; pueden usarse clases probadas en forma minuciosa sin conocer los detalles internos de cómo está construida la clase. En esta sección, usaremos otra clase proporcionada por la biblioteca estándar de C++, la clase `string`. Sin embargo, en este caso, se crearán objetos de la clase antes de usarlos, en lugar de usar un objeto existente, como `cout`.

Una clase es un tipo de datos creado por un usuario. Como los tipos de datos integrados, una clase define un conjunto válido de valores de datos y un conjunto de operaciones que pueden utilizarse en ellos. La diferencia entre una clase creada por un usuario y un tipo integrado es la forma en que está construida la clase. Un tipo de datos integrado es proporcionado como una parte integral del compilador, y una clase es construida por un programador usando código C++. Aparte de eso y de la terminología usada, los dos tipos se utilizan en forma muy parecida. La diferencia fundamental en la terminología es que las áreas de almacenamiento para los tipos integrados se conocen como variables, mientras las áreas de almacenamiento declaradas para una clase se conocen como objetos.

Los valores permitidos por la clase `string` se conocen como literales de cadena. Un literal de cadena es cualquier secuencia de caracteres encerrada entre comillas. Una literal de cadena también se conoce como un valor de cadena, una constante de cadena y, de manera más convencional, una cadena. Son ejemplos de cadenas `"Esta es una cadena"`, `"¡Hola mundo!"` y `"xyz 123 *!#@&"`. Las comillas indican los puntos inicial y final de la cadena y nunca se almacenan con ella.

La figura 7.1 muestra la representación de programación de la cadena `Hielo` siempre que se crea esta cadena como un objeto de la clase `string.` Por convención, al primer carácter en una cadena siempre se le asigna una posición 0. Este valor de posición también se conoce como valor índice del carácter y valor de compensación.

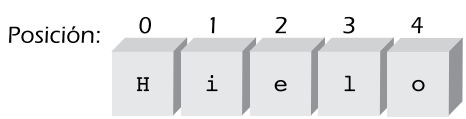


Figura 7.1 El almacenamiento de una cadena como una secuencia de caracteres.

Funciones de la clase string

La clase string proporciona diversas funciones para declarar, crear e inicializar una cadena. En las versiones anteriores de C++, el proceso de crear un objeto nuevo se conoce como instanciar un objeto, lo cual en este caso se convierte en instanciar un objeto de cadena, o crear una cadena, para abreviar. La tabla 7.2 enlista las funciones proporcionadas por la clase string para crear e inicializar un objeto de cadena. En terminología de clases, las funciones se conocen de manera formal como métodos, y los métodos que llevan a cabo esta tarea se denominan métodos constructores, o constructores, para abreviar.

Tabla 7.2 Constructores de clase string (requiere el archivo de encabezado string)

Constructor	Descripción	Ejemplos

`string nombreObjeto = valor`	Crea e inicializa un objeto de cadena a un valor que puede ser un literal de cadena, un objeto de cadena declarado con anterioridad o una expresión que contiene literales de cadena y objetos de cadena	`string str1 = "Buenos dias";`
		`string str2 = str1;`
		`string str3 = str1 + str2;`

`string nombreObjeto (valorCadena)`	Produce la misma inicialización que el anterior	`string str1 ("Hot");` `string str1 (str1 + " Dog");`
`string nombreObjeto (str, n)`	Crea e inicializa un objeto de cadena con una subcadena del objeto de cadena `str`, iniciando en la posición índice `n` de `str`	`string str1(str2, 5)` Si `str2` contiene la cadena `Buenos dias`, entonces `str1` se convierte en la cadena `dias`
`string nombreObjeto (str, n, p)`	Crea e inicializa un objeto de cadena con una subcadena del objeto de cadena `str`, iniciando en la posición índice `n` de `str` y contiene `p` caracteres	`string str1(str2, 5,2)` Si `str2` contiene la cadena `Buenos dias`, entonces `str1` se vuelve la cadena `di`
`string nombreObjeto (n, char)`	Crea e inicializa un objeto de cadena con `n` copias de `char`	`string str1(5,'*')` Esto hace a `str1 = "*****"`
`string nombreObjeto;`	Crea e inicializa un objeto de cadena para representar una secuencia de caracteres vacía (igual a la cadena `nombreObjeto = "";` el largo de la cadena es `0`)	`string mensaje;`

El programa 7.3 ilustra ejemplos de cada uno de los métodos constructores proporcionados por la clase `string`.

Programa 7.3

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

int main()
{
  string str1; // una cadena vacía
  string str2("Buenos dias");
  string str3 = "Hot Dog";
  string str4(str3);
  string str5(str4, 4);
  string str6 = "lineal";
  string str7(str6, 3, 3);

  cout << "str1 es: " << str1 << endl;
  cout << "str2 es: " << str2 << endl;
  cout << "str3 es: " << str3 << endl;
  cout << "str4 es: " << str4 << endl;
  cout << "str5 es: " << str5 << endl;
  cout << "str6 es: " << str6 << endl;
  cout << "str7 es: " << str7 << endl;

  return 0;
}
```

Aquí está la salida creada por el programa 7.3:

```
str1 es:
str2 es: Buenos dias
str3 es: Hot Dog
str4 es: Hot Dog
str5 es: Dog
str6 es: lineal
str7 es: eal
```

Aunque esta salida es sencilla, `str1` es una cadena vacía consistente de ningún carácter; debido a que al primer carácter en una cadena se le designa como posición cero, no uno, la posición de carácter de la `D` en la cadena `Hot Dog` se localiza en la posición cuatro, lo cual se muestra en la figura 7.2.

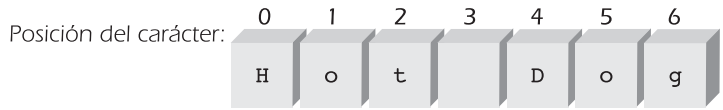


Figura 7.2 *Las posiciones de carácter de la cadena* `Hot Dog.`

Entrada y salida de `string`

Además de que una cadena se inicializa usando los métodos constructores enlistados en la tabla 7.2, las cadenas pueden ser introducidas desde el teclado y desplegarse en la pantalla. La tabla 7.3 enlista los métodos y objetos básicos que pueden usarse como entradas y salidas de valores de cadena.

Tabla 7.3 Rutinas de entrada y salida de la clase `string`

Rutina C++	Descripción
`cout`	Salida de pantalla de propósito general
`cin`	Entrada terminal de propósito general que deja de leer cuando encuentra un espacio en blanco
`getline(cin, strObj)`	Entrada terminal de propósito general que introduce todos los caracteres capturados en la cadena, `strObj`, y deja de aceptar caracteres cuando recibe un carácter de línea nueva `(\n)`

Además de los flujos `cout` y `cin` estándar, la clase `string` proporciona el método `getline()` para la entrada de cadenas. Por ejemplo, la expresión `getline(cin, mensaje)` aceptará y almacenará en forma continua caracteres mecanografiados en la terminal hasta que se oprima la tecla Entrar. Oprimir la tecla Entrar en la terminal genera un carácter de línea nueva, `'\n'`, el cual es interpretado por `getline()` como la entrada fin de línea. Todos los caracteres encontrados por `getline()`, excepto el carácter de línea nueva, son almacenados en la cadena, mensaje, como se ilustra en la figura 7.3.

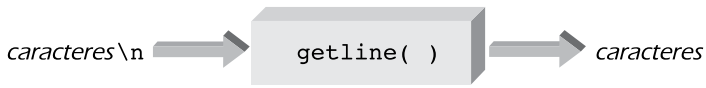


Figura 7.3 *Introducción de una cadena con* `getline()`.

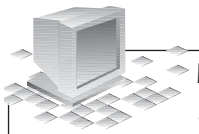
El programa 7.4 ilustra el uso del método `getline()` y el flujo `cout` para la entrada y salida de una cadena, respectivamente, que es introducida en la terminal del usuario.

La siguiente es una muestra de ejecución del programa 7.4:

```
Introduzca una cadena:
Esta es una entrada de prueba de una cadena de caracteres.
La cadena que se acaba de introducir es:
Esta es una entrada de prueba de una cadena de caracteres.
```

Aunque el objeto de flujo `cout` se usa en el programa 7.4 para la salida de cadena, por lo general el objeto de entrada de flujo `cin` no puede usarse en lugar de `getline()` para la entrada de cadena. Esto se debe a que el objeto `cin` lee un conjunto de caracteres hasta un espacio en blanco o un carácter de línea nueva. Por tanto, intentar introducir los ca-

racteres `Esta es una cadena` usando la instrucción `cin >> mensaje;` sólo produce que la palabra `Esta` se asigne a `mensaje`. El hecho que un espacio en blanco termine una operación de extracción `cin` restringe la utilidad del objeto `cin` para introducir datos de cadena y es la razón para usar `getline()`.



Programa 7.4

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

int main()
{
  string mensaje;     // declara un objeto string

  cout << "Introduzca una cadena:\n";

  getline(cin, message);

  cout << "La cadena que se acaba de introducir es:\n"
       << message << endl;


  return 0;
}
```

En su forma más general, el método `getline()` tiene la sintaxis

```
getline(cin, strObj, carácter-de-terminación)
```

donde *`strObj`* es el nombre de una variable de cadena y *`carácter-de-terminación`* es una constante, o variable, de carácter opcional que especifica el carácter de terminación. Por ejemplo, la expresión `getline(cin, mensaje, '!')` aceptará todos los caracteres introducidos en el teclado, incluyendo un carácter de línea nueva, hasta que se introduzca un signo de exclamación. El signo de exclamación no se almacenará como parte de la cadena.

Si se omite el tercer argumento opcional cuando se llama a `getline()`, el carácter de terminación por omisión es el carácter de línea nueva (`'\n'`). Por tanto, la instrucción `getline(cin, mensaje, '\n');` puede utilizarse en lugar de la instrucción `getline (cin, mensaje);`. Ambas instrucciones dejan de leer caracteres cuando se oprime la tecla Entrar. Para todos los programas usados desde este punto en adelante, se supone que la entrada es terminada al oprimir la tecla Entrar, lo cual genera un carácter de línea nueva. Así, el tercer argumento opcional transmitido a `getline()`, el cual es el carácter de terminación, se omitirá.

Precaución: El carácter de nueva línea fantasma

Al parecer pueden obtenerse resultados no deseados cuando se usan juntos el flujo de entrada `cin` y el método `getline()` para aceptar datos o cuando se utiliza el flujo de entrada `cin`, solo, para aceptar caracteres individuales. Para observar cómo puede ocurrir esto, considere el programa 7.5, el cual usa `cin` para aceptar un número entero introducido en el teclado, almacenándolo en la variable `valor`, y es seguido por una llamada al método `getline()`.

Programa 7.5

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

int main()
{
  int valor;
  string mensaje;

  cout << "Introduzca un número: ";
  cin >> valor;
  cout << "El número introducido es:\n"
       << valor << endl;

  cout << "Introduzca texto:\n";
  getline(cin, mensaje);
  cout << "El texto introducido es:\n"
       << mensaje << endl;
  cout << int(message.length());


  return 0;
}
```

Punto de información

Los tipos de datos `string` y `char`

Una cadena puede consistir de cero, uno o más caracteres. Cuando la cadena no tiene caracteres, se dice que es una cadena vacía con una longitud de cero. Una cadena con un solo carácter, como `"a"`, es una cadena de longitud uno y se almacena de manera diferente que un tipo de datos `char`, como `'a'`. Sin embargo, para muchos propósitos prácticos, una cadena de longitud uno y un `char` responden de la misma manera; por ejemplo, `cout >> "\n"` y `cout >> '\n'` producen una línea nueva en la pantalla. Es importante entender que son tipos de datos diferentes; por ejemplo, ambas declaraciones

```
string s1 = 'a';  // INICIALIZACION INVALIDA
char key = "\n"; // INICIALIZACION INVALIDA
```

producen un error de compilador debido a que intentan inicializar un tipo de datos con valores literales de otro tipo.

Cuando se ejecuta el programa 7.5, el número introducido en respuesta al indicador `Introduzca un número:` se almacena en la variable `valor`. En este punto, todo parece funcionar bien. Hay que observar, sin embargo, que al introducir un número, se introduce un número y se oprime la tecla Entrar. En casi todos los sistemas de cómputo, estos datos introducidos se almacenan en un área de contención temporal llamada búfer inmediatamente después que se han introducido los caracteres, como se ilustra en la figura 7.4.

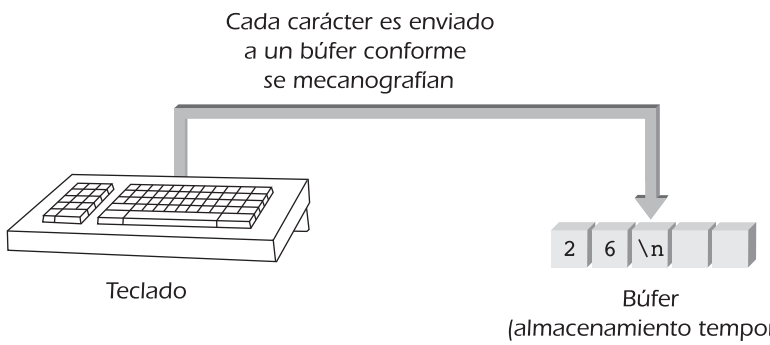


Figura 7.4 *Los caracteres mecanografiados en el teclado se almacenan primero en un búfer.*

El flujo de entrada `cin` en el programa 7.5 acepta primero el número introducido pero deja `'\n'` en el búfer. La siguiente instrucción de entrada, la cual es una llamada a `getline()`, recoge el código para la tecla Entrar como el siguiente carácter y termina cualquier entrada posterior. A continuación se presenta una muestra de ejecución para el programa 7.5:

```
Introduzca un número: 26
El número introducido es 26
Introduzca texto:
El texto introducido es
```

En esta salida, no se acepta ningún texto en respuesta al indicador `Introduzca texto:`. No ocurre ningún texto porque, después que ha sido aceptado el número 26 por el programa, el código para la tecla Entrar, la cual es una secuencia de escape de línea nueva, permanece en el búfer y es recogida e interpretada por el método `getline()` como el final de su entrada. Esto ocurrirá ya sea que sea aceptado por `cin` un número entero, como en el programa 7.5, una cadena o cualquier otra entrada y luego sea seguido por una llamada al método `getline()`.

Hay tres soluciones separadas para este problema de la tecla Entrar "fantasma":

- No mezclar entradas `cin` con `getline()` en el mismo programa.
- Hacer que a la entrada `cin` siga la llamada a `cin.ignore()`.
- Aceptar la tecla Entrar en una variable de carácter y luego ignorarla.

La solución más usada es la primera. Sin embargo, todas las soluciones se basan en el hecho que la tecla Entrar es una entrada de carácter legítima y debe reconocerse como tal. Encontrará este problema de nuevo cuando consideremos aceptar tipos de datos `char` en la siguiente sección.

Procesamiento de cadena

Las cadenas pueden manipularse usando métodos de clase `string` o el método de un carácter a la vez descrito en la siguiente sección. La tabla 7.4 enlista los métodos de clase `string` más usados. Éstos incluyen los métodos `accessor` y `mutator` más los métodos y funciones de operador que usan los operadores aritméticos y de comparación estándares.

Tabla 7.4 Los métodos de procesamiento de la clase `string` (requiere el archivo de encabezado `string`)

Método/Operación	Descripción	Ejemplo
`int length()`	Devuelve la longitud de la cadena implícita	`string.length()`
`int size()`	Igual que la anterior	`string.size()`
`at(int index)`	Devuelve el carácter en el índice especificado y lanza una excepción si el índice es inexistente	`string.at(4)`
`int compare(string)`	Compara dos cadenas; devuelve un valor negativo si la cadena implicada es menor que `str`, cero si son iguales y un valor positivo si la cadena implicada es mayor que `str`	`string1.compare(string2)`
`c_str()`	Devuelve la cadena como una cadena C terminada en `null`	`string1.c_str()`

Tabla 7.4 Los métodos de procesamiento de la clase `string` (requiere el archivo de encabezado `string`) (continuación)

Método/Operación	Descripción	Ejemplo
`bool empty`	Devuelve `verdadero` si la cadena implicada está vacía; de lo contrario, devuelve `falso`	`string1.empty()`
`erase(ind,n);`	Elimina `n` caracteres de la cadena implicada, empezando en el índice `ind`	`string1.erase(2,3)`
`erase(ind)`	Elimina todos los caracteres de la cadena implicada, empezando desde el índice `ind` hasta el final de la cadena. La longitud de la cadena restante se convierte en `ind`	`string1.erase(4)`
`int find(str)`	Devuelve el índice de la primera ocurrencia de `str` dentro del objeto implicado	`string1.find("el")`
`int find(str, ind)`	Devuelve el índice de la primera ocurrencia de `str` dentro del objeto implicado, con la búsqueda comenzando en el índice `ind`	`string1.find("el", 5)`
`int find_first_of(str, ind)`	Devuelve el índice de la primera ocurrencia de cualquier carácter en `str` dentro del objeto implicado, con la búsqueda iniciando en el índice `ind`	`string1.find_first_of("lt", 6)`
`int find_first_not_of(str, ind)`	Devuelve el índice de la primera ocurrencia de cualquier carácter que no está en `str` dentro del objeto implicado, con la búsqueda comenzando en el índice `ind`	`string1.find_first_not_of("lt ,6)`
`void insert(ind, str)`	Inserta la cadena `str` en la cadena implicada, comenzando en el índice `ind`	`string.insert(4, "ahí")`

Tabla 7.4 Los métodos de procesamiento de la clase `string` (requiere el archivo de encabezado `string`) (continuación)

Método/Operación	Descripción	Ejemplo
`void replace(ind, n, str)`	Elimina `n` caracteres en el objeto implicado, comenzando en la posición del índice `ind` e insertando la cadena `str` en la posición del índice `ind`	`string1.replace(2,4,"bien")`
`string substr(ind,n)`	Devuelve una cadena consistente de `n` caracteres extraídos de la cadena implicada empezando en el índice `ind`. Si `n` es mayor que el número restante de caracteres, se usa el resto de la cadena implicada	`string2 = string1.substr(0,10)`
`void swap(str)`	Intercambia caracteres en `str` con el objeto implicado	`string1.swap(string2)`
`[ind]`	Devuelve el carácter en el índice `x`, sin verificar si `ind` es un índice válido	`string1[5]`
`=`	Asignación (también convierte una cadena C en una cadena)	`string1 = string`
`+`	Concatena dos cadenas	`string1 + string2`
`+=`	Concatenación y asignación	`string2 += string1`

`==  !=`
`<  <=`
`>  >=`
	Operadores relacionales. Devuelve `verdadero` si la relación se satisface; de lo contrario devuelve `falso`	`string1 == string2`
		`string1 <= string2`
		`string1 > string2`

El método más usado en la tabla 7.4 es el método `length()`. Éste devuelve el número de caracteres en la cadena, lo cual se conoce como la longitud de la cadena. Por ejemplo, el valor devuelto por la llamada al método `"¡Hola mundo!".length()` es 12. Como siempre, las comillas que encierran a un valor de cadena no se consideran parte de ésta. Del mismo modo, si la cadena referenciada por string1 contiene el valor `"Ten un buen dia."`, el valor devuelto por la llamada `string1.length()` es 16.

Puede compararse la igualdad de dos expresiones de cadena usando los operadores relacionales estándar. Cada carácter en una cadena es almacenado en binario usando el código ASCII o UNICODE. Aunque estos códigos son diferentes, tienen algunas características en común. En cada uno de ellos, un blanco precede (es menor que) todas las letras y números; las letras del alfabeto se almacenan en orden de la A a la Z; y los dígitos se almacenan en orden del 0 al 9. En ambos códigos de caracteres, los dígitos van antes (es decir, son menores que) los caracteres en mayúsculas, los cuales son seguidos por los caracteres en minúsculas. Por tanto, los caracteres en mayúsculas son matemáticamente menores que los caracteres en minúsculas. Cuando se comparan dos cadenas, sus caracteres individuales se comparan un par a la vez (ambos caracteres primeros, luego ambos caracteres segundos, etc.). Si no se encuentran diferencias, las cadenas son iguales; si se encuentra una diferencia, la cadena con el primer carácter en minúscula se considera la cadena más pequeña.

- "Hola" es mayor que "Adiós" porque la primera 'H' en Hola es mayor que la primera 'A' en Adiós.
- "Hola" es menor que "hola" porque la primera 'H' en Hola es menor que la primera 'h' en hola.
- "SUÁREZ" es mayor que "JÍMENEZ" porque la primera 'S' en SUÁREZ es mayor que la primera 'J' en JÍMENEZ.
- "123" es mayor que "1227" porque el tercer carácter, '3', en 123 es mayor que el tercer carácter, '2', en 1227.
- "Bejuco" es mayor que "Bebé" porque el tercer carácter, 'j', en Bejuco es mayor que el tercer carácter, 'b', en Bebé.

El programa 7.6 usa `length()` y varias expresiones relacionales dentro del contexto de un programa completo.

Programa 7.6

```
#include <iostream>

#include <string>
using namespace std;

int main()
{
  string string1 = "Hielo";
  string string2 = "Hielo polar";

  cout << "string1 es la cadena: " << string1 << endl;
  cout << "El número de caracteres en string1 es " << int(string1.length())
       << endl << endl;

  cout << "string2 es la cadena: " << string2 << endl;
  cout << "El número de caracteres en string2 es: " << int(string2.length())
       << endl << endl;

  if (string1 < string2)
    cout << string1 << " es menor que " << string2 << endl << endl;
  else if (string1 == string2)
    cout << string1 << " es igual a " << string2 << endl << endl;
  else
    cout << string1 << " es mayor que " << string2 << endl << endl;

  string1 = string1 + " polar helado";
  cout << "Después de la concatenación, string1 contiene los caracteres: " << string1 << endl;
  cout << "La longitud de esta cadena es " << int(string1.length()) << endl;

  return 0;
}
```

A continuación hay una muestra de la salida producida por el programa 7.6:

```
string1 es la cadena: Hielo
El numero de caracteres en string1 es 5
```

```
string2 es la cadena: Hielo polar
El número de caracteres en string2 es: 11

Hielo es menor que Hielo polar

Después de la concatenación, string1 contiene los caracteres: Hielo po-
lar helado
La longitud de esta cadena es 18
```

Cuando revise esta salida, refiérase a la figura 7.5, la cual muestra cómo se almacenan en la memoria los caracteres en `string1` y `string2.` La longitud de cada cadena se refiere al número total de caracteres en la cadena, y el primer carácter en cada cadena se localiza en la posición índice 0. Por tanto, la longitud de una cadena siempre es uno más que el número índice de la última posición del carácter en la cadena.

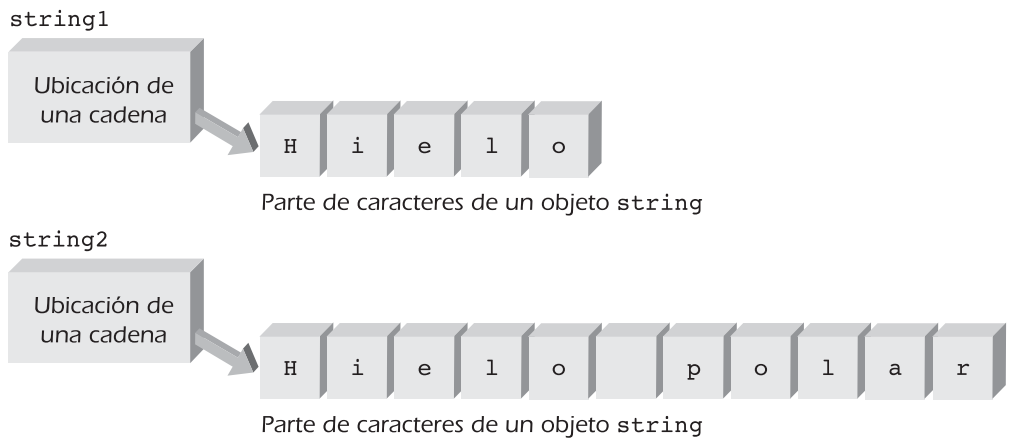
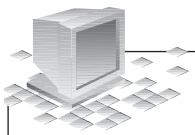


Figura 7.5 *Las cadenas iniciales usadas en el programa 7.6.*

Aunque usará fundamentalmante el operador de concatenación y el método `length()`, habrá ocasiones en que encontrará útiles los otros métodos de cadena, los cuales se describen en la tabla 7.4. Uno de los más útiles de éstos es el método `at()`, el cual le permite recuperar caracteres individuales en una cadena. El programa 7.7 utiliza este método para seleccionar un carácter a la vez de la cadena, comenzando en la posición cero de la cadena y terminando en el índice del último carácter en ella. Este último valor índice siempre es uno menos que el número de caracteres en la cadena (es decir, la longitud de la cadena).



Programa 7.7

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

int main()
{

  string str = "Contar el número de vocales";
  int i, numChars;
  int cuentaVocales = 0;

  cout << "La cadena: " << str << endl;

  numChars = int(str.length());
  for (i = 0; i < numChars; i++)
  {
    sswitch(str.at(i)    // aqui es donde se recupera un caracter
    {
      caso 'a':
      caso 'e':
      caso 'i':
      caso 'o':
      caso 'u':
        cuentaVocales++;
    }
  }
  cout << "tiene " << cuentaVocales << " vocales." << endl;


  return 0;
}
```

La expresión `str.at(i)` en la instrucción `switch` anterior recupera el carácter en la posición `i` en la cadena. Este carácter se compara luego con cinco valores de carácter diferentes. La instrucción `switch` utiliza el hecho que los casos seleccionados son "ejecutados" en ausencia de instrucciones break. Por tanto, todos los casos seleccionados producen un incremento en `cuentaVocales.` La salida desplegada por el programa 7.7 es la siguiente:

```
La cadena: Contar el número de vocales
tiene 9 vocales.
```

Como ejemplo para insertar y reemplazar caracteres en una cadena, usando los métodos enlistados en la tabla 7.4, suponga que comienza con una cadena creada por la siguiente instrucción:

```
string str = "Debe pensar si";
```

La figura 7.6 ilustra cómo se almacena esta cadena en el búfer creado para ella. Como se indica, la longitud inicial de la cadena es de 14 caracteres.

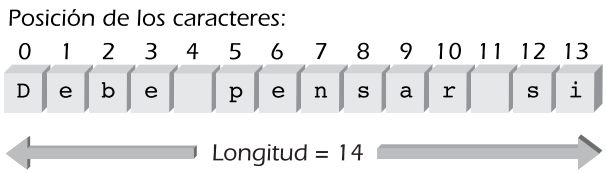


Figura 7.6 *Almacenamiento inicial de un objeto de cadena.*

Ahora suponga que se ejecuta la siguiente instrucción:

```
str.insert(4, " sentir");
```

Esta instrucción causa que se inserten los siete caracteres designados, comenzando con un espacio en blanco, a partir de la posición índice 4, en la cadena existente. La cadena resultante, después de la inserción, es como se muestra en la figura 7.7.

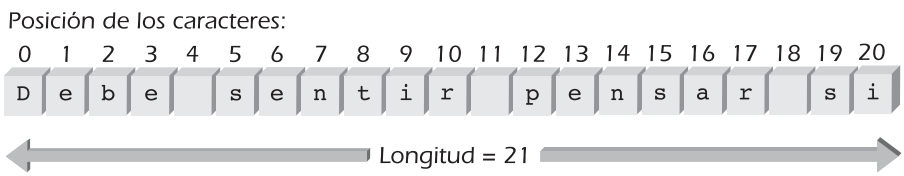


Figura 7.7 *La cadena después de la inserción.*

Si ahora se ejecuta la instrucción `str.replace(12, 6, "tu");`, los caracteres existentes en las posiciones de índice 12 a 17 se eliminarán y los dos caracteres `tu` se insertarán empezando en la posición de índice 12. Por tanto, el efecto neto del reemplazo es como se muestra en la figura 7.8. El número de caracteres de reemplazo, que en este caso particular son dos, puede ser menor que, igual a o mayor que los caracteres que se están reemplazando, los cuales en este caso son seis.

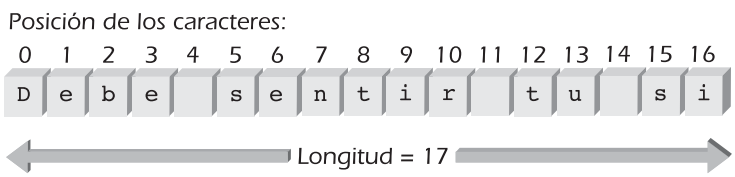


Figura 7.8 *La cadena después del reemplazo.*

Por último, si se anexa la cadena `"podrias"` a la cadena mostrada en la figura 7.8 usando el operador de concatenación, `+`, se obtiene la cadena ilustrada en la figura 7.9.

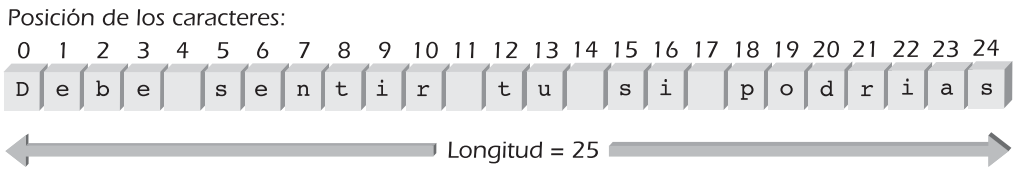


Figura 7.9 *La cadena después de la anexión.*

El programa 7.8 ilustra el uso de las instrucciones dentro del contexto de un programa completo.

Programa 7.8

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

int main()
{
  string str = "Debe pensar si";

  cout << "La cadena original es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

  // insertar caracteres
  str.insert(4, " sentir");
  cout << "La cadena, después de la inserción, es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

  // reemplazar caracteres
  str.replace(12, 6, "tu");
  cout << "La cadena, después del reemplazo, es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

  // anexar caracteres
  str = str + " podrias";
  cout << "La cadena, después de la anexión, es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

   return 0;
}
```

La siguiente salida producida por el programa 7.8 corresponde a las cadenas mostradas en las figuras 7.6 a 7.9:

```
La cadena original es: Debe pensar si
  y tiene 14 caracteres.
La cadena, después de la insercion, es: Debe sentir pensar si
  y tiene 21 caracteres.
La cadena, después del reemplazo, es: Debe sentir tu si
  y tiene 17 caracteres.
La cadena, después de la anexion, es: Debe sentir tu si
podrias
  y tiene 25 caracteres.
```

De los métodos de cadena restantes enlistados en la tabla 7.4, los más usados son aquellos que localizan caracteres específicos en una cadena y crean subcadenas. El programa 7.9 presenta ejemplos de la forma en que se utilizan algunos de estos métodos.

Programa 7.9

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

int main()
{

  string string1 = "LINEAR PROGRAMMING THEORY";
  string s1, s2, s3;
  int j, k;

  cout << "La cadena original es " << string1 << endl;

  j = int(string1.find('I'));
  cout << " La primera posición de una 'I' es " << j << endl;

  k = int(string1.find('I', (j+1)));
  cout << " La siguiente posición de una 'I' es " << k << endl;

  j = int(string1.find("THEORY"));
  cout << " La primera ubicación de \"THEORY\" es " << j << end;

  k = int(string1.find("ING"));
  cout << " El primer índice de \"ING\" es " << k << endl;
```

(Continúa)

(Continuación)

```
    s1 = string1.substr(2,5);
    s2 = string1.substr(19,3);
    s3 = string1.substr(6,8);

    cout << s1 + s2 + s3 << endl;


    return 0;
}
```

Aquí está la salida producida por el programa 7.9:

```
La cadena original es LINEAR PROGRAMMING THEORY
  La primera posicion de una 'I' es 1
  La siguiente posicion de una 'I' es 15
  La primera ubicacion de "THEORY" es 19
  El primer indice de "ING" es 15
NEAR THE PROGRAM
```

El punto principal ilustrado en el programa 7.9 es que pueden localizarse y extraerse de una cadena caracteres individuales y secuencias de caracteres.

Ejercicios 7.2

1. Introduzca y ejecute el programa 7.4.
2. Determine el valor de `text.at(0)`, `text.at(3)` y `text.at(10)`, asumiendo que el texto es, individualmente, cada una de las siguientes cadenas:
   a. ahora es el momento
   b. el mapache rocky le da la bienvenida
   c. Felices fiestas
   d. El buen barco
3. Introduzca y ejecute el programa 7.7.
4. Modifique el programa 7.7 para contar y desplegar los números individuales de cada vocal contenida en la cadena.
5. Modifique el programa 7.7 para desplegar el número de vocales en una cadena introducida por el usuario.
6. Usando el método `at()`, escriba un programa en C++ que lea en una cadena usando `getline()` y luego despliegue la cadena en orden inverso. (*Sugerencia:* Una vez que se ha introducido y guardado la cadena, recupere y despliegue caracteres empezando por el final de la cadena.)
7. Escriba un programa en C++ que acepte tanto una cadena como un solo carácter del usuario. El programa deberá determinar cuántas veces está contenido el carácter en la cadena. (*Sugerencia:* Busque la cadena usando el método `find(str,`

ind). Este método deberá utilizarse en un ciclo que comience el valor índice en cero y luego lo cambie al valor uno pasado de donde se encontró por última vez el carácter.)

8. Introduzca y ejecute el programa 7.8.
9. Introduzca y ejecute el programa 7.9.
10. Escriba un programa en C++ que acepte una cadena del usuario y luego reemplace todas las ocurrencias de la letra e con la letra x.
11. Modifique el programa escrito para el ejercicio 10 para buscar la primera ocurrencia de una secuencia de caracteres introducida por el usuario y reemplazar esta secuencia, cuando se encuentre en la cadena, con un segundo conjunto de una secuencia introducida por el usuario. Por ejemplo, si la cadena introducida es La `figura 4-4 ilustra la salida del programa 4-2` y el usuario introduce que 4- sea reemplazado por 3−, la cadena resultante será La `figura 3-4 ilustra la salida del programa 4-2`. (Sólo se ha cambiado la primera ocurrencia de lo buscado en la secuencia.)
12. Modifique el programa escrito para el ejercicio 11 para reemplazar todas las ocurrencias de la secuencia de caracteres designada con la nueva secuencia de caracteres. Por ejemplo, si la cadena introducida es La `figura 4-4 ilustra la salida del programa 4-2` y el usuario introduce que 4− ha de ser reemplazado por 3−, la cadena resultante será La `figura 3-4 ilustra la salida del programa 3-2.`

7.3 MÉTODOS DE MANIPULACIÓN DE CARACTERES

Además de los métodos `string` proporcionados por la clase string, el lenguaje C++ proporciona diversas funciones útiles de la clase `character.` Estas funciones se muestran en la tabla 7.5. Las declaraciones de función (prototipos) para cada una de estas rutinas están contenidas en el archivo de encabezado `string` y `cctype`, los cuales deben incluirse en cualquier programa que use estas funciones.

Tabla 7.5 Funciones de biblioteca de carácter (requieren el archivo de encabezado string o `cctype`)

Prototipo de la función	Descripción	Ejemplo
`int isalpha(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra; de lo contrario, devuelve `falso` (número entero cero)	`isalpha('a')`

`int isalnum(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra o un dígito; de lo contrario, devuelve `falso` (número entero cero)	`char key;`
		`cin >> key;`
		`isalnum(key);`

Tabla 7.5 Funciones de biblioteca de carácter (requieren el archivo de encabezado string o `cctype`) (continuación)

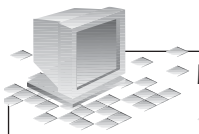
Prototipo de la función	Descripción	Ejemplo
`int isupper(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra mayúscula; de lo contrario, devuelve `falso` (número entero cero)	`isupper('a')`
`int islower(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra minúscula; de lo contrario, devuelve `falso` (número entero cero)	`islower('a')`
`int isdigit(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un dígito (0 a 9); de lo contrario, devuelve `falso` (número entero cero)	`isdigit('a')`
`int isascii(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter ASCII; de lo contrario, devuelve `falso` (número entero cero)	`isascii('a')`
`int isspace(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un espacio; de lo contrario, devuelve `falso` (número entero cero)	`isspace(' ')`
`int isprint(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter imprimible; de lo contrario, devuelve `falso` (número entero cero)	`isprint('a')`
`int isctrl(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter de control; de lo contrario, devuelve `falso` (número entero cero)	`isctrl('a')`
`int ispucnt(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter de puntuación; de lo contrario, devuelve `falso` (número entero cero)	`ispucnt('!')`
`int isgraph(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter imprimible diferente a un espacio en blanco; de lo contrario, devuelve `falso` (número entero cero)	`isgraph(' ')`
`int toupper(charExp)`	Devuelve el equivalente en mayúscula si `charExp` evalúa un carácter en minúscula; de lo contrario, devuelve el código de carácter sin modificación	`toupper('a')`
`int tolower(charExp)`	Devuelve el equivalente en minúscula si `charExp` evalúa un carácter en mayúscula; de lo contrario, devuelve el código de carácter sin modificación	`tolower('A')`

Debido a que todas las funciones `istype()` enlistadas en la tabla 7.5 devuelven un número entero diferente de cero (el cual es interpretado como un valor booleano verdadero) cuando el carácter satisface la condición deseada y un número entero cero (o valor boo-

leano `falso`) cuando la condición no se cumple, por lo general estas funciones se usan en forma directa dentro de una instrucción `if`. Por ejemplo, considérese el siguiente segmento de código, el cual supone que `ch` es una variable de carácter:

```
if(isdigit(ch))
  cout << "El carácter que se acaba de introducir es un dígito" << endl;
else if(ispunct(ch))
  cout << "El carácter que se acaba de introducir es un signo de pun-
  tuación" << endl;
```

En este ejemplo, si `ch` contiene un carácter de dígito, se ejecuta la primera instrucción `cout`; si el carácter es una letra, se ejecuta la segunda instrucción `cout`. En ambos casos, sin embargo, el carácter que se va a comprobar es incluido como un argumento para el método apropiado. El programa 7.10 ilustra este tipo de código dentro de un programa que cuenta el número de letras, dígitos y otros caracteres en una cadena. Los caracteres individuales que se van a comprobar se obtienen usando el método `at()` de la clase `string`. En el programa 7.10, este método se usa en un ciclo `for` que circula por la cadena desde el primer carácter hasta el último.



Programa 7.10

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

int main()
{
    sstring str = "Esta -123/ es 567 A ?<6245> foca!";
    char siguienteCar;
    int i;
    int numLetras = 0, numDigitos = 0, numOtros = 0;

    cout << "La cadena original es: " << str
         << "\nEsta cadena contiene " << int(str.length())
            << " caracteres," << " la cual consiste de" << endl;

    // verifica cada carácter en la cadena
    for (i = 0; i < int(str.length()); i++)
    {
      siguienteCar = str.at(i); // obtiene un carácter
      if (isalpha(siguienteCar))
        numLetras++;
      else if (isdigit(siguienteCar))
        numDigitos++;
```

(Continúa)

(Continuación)

```
        else
          numOtros++;
      }

      cout << "      " <<  numLetras << " letras" << endl;
      cout << "      " <<  numDigitos << " digitos" << endl;
      cout << "      " <<  numOtros << " otros caracteres." << endl;

    cin.ignore();
    return 0;
}
```

La salida producida por el programa 7.10 es la siguiente:

```
La cadena original es: Esta -123/ es 567 A ?<6245> foca!
Esta cadena contiene 33 caracteres, los cuales consisten de
     11 letras
     10 digítos
     12 otros caracteres.
```

Como lo indica esta salida, cada uno de los 33 caracteres en la cadena ha sido clasificado en forma correcta como una letra, dígito u otro carácter.

Por lo general, como en el programa 7.10, cada una de las funciones en la tabla 7.5 se utiliza carácter por carácter en cada carácter en una cadena. Esto se ilustra de nuevo en el programa 7.11, donde cada carácter en minúsculas en la cadena es convertido a su equivalente en mayúsculas usando la función `toupper()`. Esta función sólo convierte las letras minúsculas, dejando todos los demás caracteres intactos.

Una muestra de la ejecución del programa 7.11 produjo la siguiente salida:

```
Mecanografíe cualquier secuencia de caracteres: esta es una prueba DE 12345.
Los caracteres recién introducidos, en mayúsculas, son: ESTA ES UNA PRUE-
BA DE 12345.
```

En el programa 7.11, ponga particular atención a la instrucción `for (i = 0; i < int(str.length()); i++)` que se utilizo para hacer un ciclo a través de cada uno de los caracteres en la cadena. Ésta es la forma típica de acceso a cada elemento en una cadena, usando el método `length()` para determinar cuando se ha alcanzado el final de la cadena. (Revise el programa 7.10 para verificar que se usa en la misma forma.) La única diferencia real es que en el programa 7.11 se tiene acceso a cada elemento usando la notación en subíndice `str[i];` en el programa 7.10 se usó el método `at()`. Aunque estas dos notaciones son intercambiables, y es cuestión de elección cuál utilizan, las dos notaciones no deberán mezclarse en el mismo programa para mantener la consistencia.

Programa 7.11

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

int main()
{
  int i;
  string str;

  cout << "Mecanografíe cualquier secuencia de caracteres: ";
  getline(cin,str);

  // hace un ciclo a través de todos los elementos de la cadena
  for (i = 0; i < int(str.length()); i++)
    str[i] = toupper(str[i]);

  cout << "Los caracteres recién introducidos, en mayúsculas, son: "
       << str << endl;

  cin.ignore();
  return 0;
}
```

E/S de caracteres

Aunque se ha utilizado `cin` y `getline()` para aceptar los datos introducidos desde el teclado de una manera más o menos parecida a un "recetario de cocina", es necesario entender qué datos se están enviando al programa y cómo debe reaccionar éste para procesar los datos. En un nivel básico, toda la entrada (al igual que la salida) se hace carácter por carácter, como se ilustra en la figura 7.10.

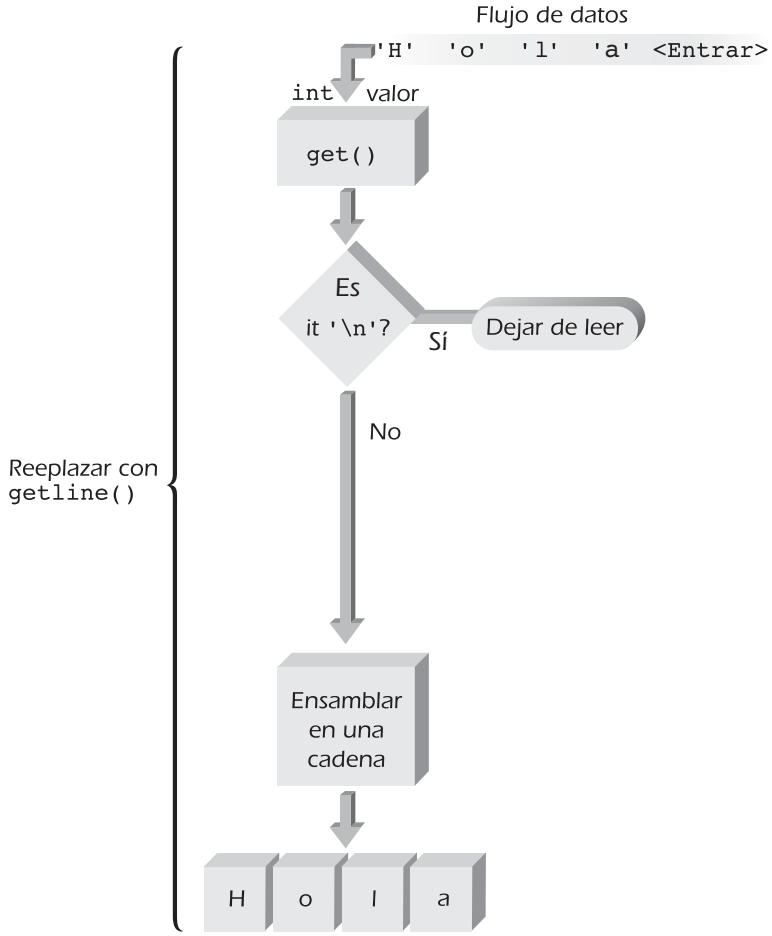


Figura 7.10 *Aceptar caracteres introducidos en el teclado.*

Como se ilustra en la figura 7.10, la introducción de cada pieza de datos, sea una cadena o un número, consiste en la mecanografía de caracteres individuales. Por ejemplo, la entrada de la cadena Hola consiste en oprimir y soltar las cuatro teclas `H`, `o`, `l`, `a` y la tecla Entrar. Del mismo modo, la salida del número `26.95` consiste en el despliegue de los cinco caracteres `2`, `6`, `.`, `9` y `5`. Aunque el programador por lo general no piensa en los datos de esta manera, el programa está restringido a esta E/S carácter por carácter, y todos los métodos y flujos E/S de nivel superior en C++ se basan en métodos de E/S de carácter de nivel inferior. Estos métodos de carácter más elemental, los cuales pueden ser usados en forma directa por un programador, se muestran en la tabla 7.6.

Punto de información

Por qué el tipo de datos `char` usa valores en números enteros

En C++, un carácter se almacena como un valor en número entero, lo cual en ocasiones es confuso para los programadores principiantes. La razón para esto es que, además de las letras y caracteres estándar en inglés, un programa necesita almacenar caracteres especiales que no tienen equivalentes imprimibles. Uno de éstos es el centinela de fin de archivo que utilizan todos los sistemas de cómputo para designar el final de un archivo de datos. Estos centinelas de fin de archivo pueden ser transmitidos desde el teclado. Por ejemplo, en sistemas basados en Unix, se genera oprimiendo el botón Ctrl y, mientras se mantiene oprimido, se oprime la tecla D; en sistemas basados en Windows, se genera oprimiendo Ctrl y, mientras se mantiene oprimido, oprimiendo la tecla Z. Ambos centinelas son almacenados como el número entero -1, el cual no tiene un valor de carácter equivalente. (Puede verificar esto desplegando el valor entero de cada carácter introducido [véase el programa 7.12] y mecanografiando la combinación Ctrl + D o la combinación Ctrl + Z, dependiendo del sistema que use.)

Al usar un valor entero de 16 bits, pueden representarse más de 64 000 caracteres diferentes. Esto proporciona suficiente almacenamiento para múltiples conjuntos de caracteres que pueden incluir árabe, chino, hebreo, japonés, ruso y casi todos los símbolos lingüísticos conocidos. Por tanto, almacenar un carácter como un valor entero tiene un valor práctico.

Una consecuencia importante de usar códigos en número entero para cadenas de caracteres es que los caracteres pueden compararse con facilidad para su ordenamiento alfabético. Por ejemplo, en tanto cada letra subsiguiente en un alfabeto tenga un valor superior que su letra precedente, la comparación de los valores de carácter se reduce a la comparación de valores numéricos. Si los caracteres se almacenan en orden numérico secuencial, esto asegura que agregar uno a una letra producirá la siguiente letra en el alfabeto.

Tabla 7.6 Métodos de E/S de carácter básicos (requieren el archivo de encabezado `cctype`)

Método	Descripción	Ejemplo
`cout.put(charExp)`	Coloca el valor de carácter de `charExp` en el flujo de salida	`cout.put('A');`
`cin.get(charVar)`	Extrae el siguiente carácter del flujo de entrada y lo asigna a la variable `charVar`	`cin.get(key);`
`cin.peek(charVar)`	Asigna el siguiente carácter del flujo de entrada a la variable `charVar` *sin extraer* el carácter del flujo	`cin.peek(nextKey);`
`cin.putback(charExp)`	Pone un valor de carácter de `charExp` de vuelta en el flujo de entrada	`cin.putback(cKey);`
`cin.ignore(n, char)`	Ignora un máximo de los siguientes `n` caracteres de entrada, hasta la detección de `char` e incluyéndola. Si no se especifican argumentos, ignora el siguiente carácter en el flujo de entrada	`cin.ignore(80,'\n');` `cin.ignore();`

Un poco de antecedentes

Una inconsistencia notacional

Todos los métodos de clase de carácter mostrados en la tabla 7.6 utilizan la notación orientada a objetos estándar en la cual el nombre del método va precedido por el nombre de un objeto, como en `cin.get()`. Éste no es el caso con el método `getline()` de la clase string el cual usa la notación `getline(cin,fstrVar)`. En esta notación, el objeto, `cin` por ejemplo, aparece como un argumento, que es la forma en la cual las funciones basadas en procedimientos transmiten las variables. En cuanto a lograr consistencia, puede esperar que `getline()` sea llamado como `cin.getline()`.

Por desgracia, la notación apropiada está en uso para un método `getline()` creado originalmente para cadenas estilo C (véase la sección 10.1); por consiguiente, fue creada una inconsistencia notacional.

La función `get()` lee el siguiente carácter en el flujo de entrada y lo asigna a la variable de carácter de la función. Por ejemplo, examine la siguiente instrucción:

```
cin.get(siguientetCar);
```

Causa que el siguiente carácter introducido en el teclado sea almacenado en la variable de carácter `siguienteCar.` Esta función es útil para introducir y verificar caracteres individuales antes que sean asignados a una cadena o tipo de datos de C++ completos.

La función de salida de carácter correspondiente a `get()` es `put()`. Esta función espera un argumento de un solo carácter y despliega el carácter que le es transmitido en la terminal. Por ejemplo, la instrucción `cout.put('A')` causa que la letra A sea desplegada en la pantalla.

De las tres últimas funciones mostradas en la tabla 7.6, la función `cin.ignore()` es la más útil. Esta función permite omitir la entrada hasta que se encuentra un carácter designado, como `'\n'`. Por ejemplo, la instrucción `cin.ignore(80, '\n')` omitirá hasta un máximo de los siguientes 80 caracteres o dejará de omitir si encuentra el carácter de línea nueva. Esta instrucción puede ser útil para omitir toda la entrada adicional en una línea, hasta un máximo de 80 caracteres, o hasta que se encuentre el final de la línea actual. La entrada comenzará con la linea siguiente.

La función `peek()` devuelve el siguiente carácter en el flujo pero no lo elimina del búfer del flujo (véase la figura 7.6). Por ejemplo, la expresión `cin.peek(siguienteCar)` devuelve el siguiente carácter introducido por medio del teclado pero lo deja en el búfer. Esto en ocasiones es útil para echar un vistazo adelante y ver cuál es el siguiente carácter, mientras se deja en su lugar para la siguiente entrada.

Por último, la función `putback()` coloca un carácter de vuelta en el flujo, de modo que será el siguiente carácter que sea leído. El argumento transmitido a `putback()` puede ser cualquier expresión de carácter que evalúe un valor de carácter legítimo y no necesite ser el último carácter introducido.

La nueva línea fantasma revisada de nuevo

Como se vio en la sección anterior, a veces se obtienen resultados aparentemente extraños cuando una entrada de flujo `cin` es seguida por una llamada al método `getline()`. Este mismo resultado puede ocurrir cuando se introducen caracteres utilizando el método de carácter `get()`. Para ver cómo puede ocurrir esto, considérese el programa 7.12, el cual usa el método `get()` para aceptar el siguiente carácter introducido en el teclado y almacenar el carácter en la variable `fkey`.

Programa 7.12

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

int main()
{
  char primera_tecla;

  cout << "Mecanografie un caracter: ";
  cin.get(primera_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(primera_tecla) << endl;

  return 0;
}
```

Cuando se ejecuta el programa 7.12, el carácter introducido en respuesta al indicador `Mecanografie un caracter:` se almacena en la variable de carácter `primera_tecla` y el código decimal para el carácter es desplegado al vaciar en forma explícita el carácter en un número entero, para forzar su despliegue como un valor entero. La siguiente muestra de ejecución ilustra esto:

```
Mecanografie un caracter: m
La tecla que se acaba de aceptar es 109
```

En este punto, todo parece funcionar, aunque podría preguntarse por qué se despliega el valor decimal de `m` en lugar del carácter en sí. La razón para esto se hará evidente.

Al mecanografiar `m`, por lo general se oprimen dos teclas, la tecla `m` y la tecla Entrar. Como en la sección anterior, estos dos caracteres se almacenan en un búfer después que se han oprimido (véase la figura 7.4).

La primera tecla oprimida, `m` en este caso, es tomada del búfer y almacenada en `primera_tecla`. Esto, sin embargo, aún deja el código para la tecla Entrar en el búfer. Por tanto, una llamada subsiguiente a `get()` para una entrada de carácter tomará de manera automática el código para la tecla Entrar como el siguiente carácter. Por ejemplo, considérese el programa 7.13.

Programa 7.13

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

int main()
{
  char primera_tecla, segunda_tecla;

  cout << "Mecanografie un caracter: ";
  cin.get(primera_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(primera_tecla) << endl;

  cout << "Mecanografie otro caracter: ";
  cin.get(segunda_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(segunda_tecla) << endl;

  return 0;
}
```

La siguiente es una muestra de la ejecución del programa 7.13.

```
Mecanografie un caracter: m
La tecla que se acaba de aceptar es 109
Mecanografie otro caracter: La tecla que se acaba de acep-
tar es 10
```

Después de introducir la letra `m` en respuesta al primer indicador, también se oprime la tecla Entrar. Desde un punto de vista de carácter, esto representa la introducción de dos caracteres distintos. El primer carácter es `m`, el cual es codificado y almacenado como el número entero 109. El segundo carácter también se almacena en el búfer con el código numérico para la tecla Entrar. La segunda llamada a `get()` toma este código de inmediato, sin esperar que se oprima cualquier tecla adicional. El último flujo `cout` despliega el código para esta tecla. La razón para desplegar el código numérico en lugar del carácter en sí se debe a que la tecla Entrar no tiene un carácter imprimible asociado con ella que pueda ser desplegado.

Recuerde que cada tecla tiene un código numérico, incluyendo las teclas Entrar, Escape y Control, así como la barra espaciadora. Estas teclas por lo general no tienen efecto cuando se introducen números debido a que los métodos de entrada las ignoran como entradas a la izquierda o a la derecha de los datos numéricos. Tampoco afectan la introducción de un solo carácter solicitado como el primer dato del usuario que se va a introducir, como en el caso del programa 7.12. Sólo cuando se solicita un carácter después que el usuario ya ha introducido algún otro dato, como en el programa 7.13, se hace evidente la tecla Entrar por lo general invisible.

En la sección 7.2, aprendió algunas formas para prevenir que la tecla Entrar sea aceptada como una entrada de carácter legítima cuando se usa el método `getline()`. Las siguientes formas pueden emplearse cuando se utiliza el método `get()` dentro de un programa:

- Hacer que la entrada `cin.get()` sea seguida de la llamada `cin.ignore()`.
- Aceptar la tecla Entrar en una variable de carácter, y luego no usarla más.

El programa 7.14 aplica la primera solución al programa 7.13. Ignorar la tecla Entrar después que se ha leído y desplegado el primer carácter vacía el búfer de la tecla Entrar y lo deja listo para almacenar el siguiente carácter válido introducido como su primer carácter.

Programa 7.14

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

int main()
{
  char primera_tecla, segunda_tecla;

  cout << "Mecanografie un caracter: ";
  cin.get(primera_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(primera_tecla) << endl;
  cin.ignore();

  cout << "Mecanografie otro caracter: ";
  cin.get(segunda_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(segunda_tecla) << endl;

  cin.ignore();
  return 0;
}
```

En el programa 7.14, hay que observar que cuando el usuario mecanografía la tecla m y oprime la tecla Entrar, la m es asignada a `primera_tecla` y el código para la tecla Entrar es ignorado. La siguiente llamada a `get()` almacena el código para la siguiente tecla oprimida en la variable `segunda_tecla`. Desde el punto de vista del usuario, la tecla Entrar no tiene efecto excepto para señalar el fin de cada entrada de carácter. La siguiente es una muestra de la ejecución del programa 7.14:

```
Mecanografie un caracter: m
La tecla que se acaba de aceptar es 109
Mecanografie otro caracter: b
La tecla que se acaba de aceptar es 98
```

Una segunda mirada a la validación de entradas del usuario

Como se mencionó durante la primera mirada a la validación de entradas del usuario (sección 3.4), los programas que responden de manera efectiva a entradas inesperadas del usuario se conocen de manera formal como programas robustos y de manera informal como programas "a prueba de balas". El código que valida las entradas del usuario y asegura que un programa no produce resultados imprevistos debido a entradas inesperadas es una señal de un programa robusto bien construido. Uno de sus trabajos, como programador, es producir tales programas. Para ver cómo pueden ocurrir resultados imprevistos, considérense los dos ejemplos de código que se presentan a continuación. Primero, supóngase que el programa contiene las siguientes instrucciones:

```
cout << "Introduzca un número entero: ";
cin >> valor;
```

Ahora supóngase que, por error, un usuario introduce los caracteres e4. En versiones anteriores de C++, esto causaría que el programa terminara en forma inesperada, o **se cayera**. Aunque todavía puede ocurrir una caída con el estándar ANSI/ISO actual (véase, por ejemplo, el ejercicio 9), no ocurrirá en este caso. Más bien, se asignará un valor entero sin sentido a la variable valor. Esto, por supuesto, invalidará los resultados obtenidos usando esta variable.

Como un segundo ejemplo, considérese el siguiente código, el cual causará que ocurra un ciclo infinito si el usuario introduce un valor no numérico (el programa puede detenerse oprimiendo el botón Ctrl y, mientras se mantiene oprimido, oprimiendo la tecla c):

```
double valor;

  do
  {
    cout << "Introduzca un número (introduzca 0 para salir): ";
    cin >> valor;

    cout << "La raiz cuadrada de este número es: " << sqrt(valor) << endl;
    }While (value !=0);
```

La técnica básica para manejar la introducción de datos inválidos y prevenir que código aparentemente inocuo, como en los dos ejemplos anteriores, produzca resultados imprevistos se conoce como ***validación de las entradas del usuario***. Esto significa validar los datos introducidos durante o después de la entrada de datos y proporcionar al usuario una forma de reintroducir los datos inválidos. La validación de las entradas del usuario es una parte esencial de cualquier programa viable desde el punto de vista comercial y, si se hace en forma correcta, protege al programa e inhibe el intento de procesar tipos de datos que pueden causar que un programa se caiga, cree ciclos infinitos o produzca otros resultados inválidos.

El elemento central en la validación de las entradas del usuario es la comprobación de cada carácter introducido para verificar que es un carácter legítimo para el tipo de datos esperado. Por ejemplo, si se requiere un número entero, los únicos caracteres aceptables son un signo de más (+) o de menos (−) a la izquierda y los dígitos 0 a 9. Estos caracteres pueden comprobarse conforme son mecanografiados, lo cual significa que la función `get()` se usa para introducir un carácter a la vez, o después que todos los caracteres son aceptados en una cadena, cada carácter de la cadena es comprobado en su validez. Una vez que han sido validados todos los caracteres introducidos, la cadena puede convertirse en el tipo de datos correcto.

Existen dos medios básicos para lograr la validez de los caracteres introducidos. Al principio de la sección 7.4 se presenta una de estas formas: la comprobación carácter por carácter. Una segunda técnica, la cual abarca un alcance más amplio de tareas de procesamiento de datos usando el manejo de excepciones, se presenta al final de la sección 7.4.

Ejercicios 7.3

1. Introduzca y ejecute el programa 7.10.
2. Introduzca y ejecute el programa 7.11.
3. Escriba un programa en C++ que cuente el número de palabras en una cadena. Una palabra se localiza siempre que se encuentra una transición de un espacio en blanco a un carácter que no está en blanco. Suponga que la cadena sólo contiene palabras separadas por espacios en blanco.
4. Genere diez números aleatorios en el rango de 0 a 129. Si el número representa un carácter imprimible, imprima el carácter con un mensaje apropiado que indique lo siguiente:

   El carácter es una letra minúscula
   El carácter es una letra mayúscula
   El carácter es un dígito
   El carácter es un espacio

   Si el carácter no es ninguno de estos datos, despliegue su valor en formato de número entero.

5. **a.** Escriba una función, `length()`, que determine y devuelva la longitud de una cadena sin usar el método de la clase de cadena `length()`.
   b. Escriba una función `main()` simple para probar la función `length()` escrita para el ejercicio 5a.
6. **a.** Escriba una función, `countlets()`, que devuelva el número de letras en una cadena transmitida como un argumento. Los dígitos, espacios, signos de puntuación, tabuladores y caracteres de línea nueva no deberán incluirse en la cuenta devuelta.
   b. Incluya el método `countlets()` escrito para el ejercicio 6a en un programa C++ ejecutable y use el programa para probar el método.
7. Escriba un programa que acepte una cadena de la consola y despliegue el equivalente hexadecimal de cada carácter en la cadena.
8. Escriba un programa en C++ que acepte una cadena de la consola y despliegue la cadena una palabra por línea.
9. En respuesta al siguiente código suponga que un usuario introduce los datos 12e4:

```
cout << "Introduzca un número entero: ";
cin >> valor;
```

   ¿Qué valor será almacenado en la variable entera `valor`?

10. a. Escriba un programa en C++ que deje de leer una línea de texto cuando se introduce un punto y despliegue el enunciado con un espaciado y uso de mayúsculas correctos. Para este programa, el espaciado correcto significa que sólo habrá un espacio entre palabras y que todas las letras deberán estar en minúsculas, excepto la primera letra. Por ejemplo, si el usuario introdujo el texto voy `a Ir AL ciNe`., el enunciado desplegado deberá ser `Voy a ir al cine.`

b. Determine qué caracteres, si es que hay alguno, no fueron desplegados en forma correcta por el programa que creó para el ejercicio 10a.

11. Escriba un programa en C++ que acepte un nombre como nombre y apellido y luego lo despliegue como apellido, nombre. Por ejemplo, si el usuario introdujo Gary Bronson, la salida deberá ser Bronson, Gary.

12. Modifique el programa escrito para el ejercicio 11 para incluir una serie de cinco nombres.

7.4 VALIDACIÓN DE DATOS DE ENTRADA

Uno de los usos principales de las cadenas en los programas es la validación de las entradas del usuario. La necesidad de validar las entradas del usuario es esencial: aunque un programa le indique al usuario que introduzca un tipo específico de datos, como un número entero, esto no asegura que el usuario lo hará. Lo que un usuario introduce, de hecho, está totalmente fuera del control del programador. Lo que sí se encuentra bajo control es cómo tratar los datos introducidos.

Por supuesto que no es bueno decirle a un usuario frustrado "El programa claramente le indicó que introdujera un número entero y usted introdujo una fecha". En cambio, los programas exitosos siempre anticipan los datos inválidos y los aíslan para que no sean aceptados y procesados. Esto se logra por lo general validando primero que los datos sean del tipo correcto; si así es, los datos son aceptados; de lo contrario, se le solicita al usuario que los reintroduzca, proporcionando una explicación de por qué son inválidos los datos introducidos.

Uno de los métodos más comunes para validar la introducción de datos numéricos es aceptar todos los números como cadenas. Cada carácter en la cadena puede ser comprobado entonces para asegurar que cumple con el tipo de datos solicitado. Después que se hace esta comprobación y los datos son verificados de acuerdo con el tipo correcto, la cadena se convierte en un valor de un número entero o de precisión doble usando las funciones de conversión enlistadas en la tabla 7.7. (Para datos aceptados usando objetos de la clase `string`, debe aplicarse el método `c_str()` a la cadena antes de invocar a la función de conversión.)

Como ejemplo, considérese la introducción de un número entero. Para ser válidos, los datos introducidos deben apegarse a las siguientes condiciones:

- Los datos deben contener al menos un carácter.
- Si el primer carácter es un signo + o −, los datos deben contener al menos un dígito.
- Sólo dígitos de 0 a 9 después del primer carácter son aceptables.

Tabla 7.7 Funciones de conversión c-string

Función	Descripción	Ejemplo
`int atoi(stringExp)`	Convierte `stringExp` en un número entero. La conversión se detiene en el primer carácter que no es un número entero.	`atoi("1234")`
`double atof(stringExp)`	Convierte `stringExp` en un número de precisión doble. La conversión se detiene en el primer carácter que no puede interpretarse como un doble.	`atof("12.34")`
`char[] itoa(integerExp)`	Convierte `integerExp` en un arreglo de caracteres. El espacio asignado para los caracteres devueltos debe ser lo bastante grande para el valor convertido.	`itoa(1234)`

La siguiente función, `isvalidInt(),` puede usarse para verificar que una cadena introducida cumple con estas condiciones. Esta función devuelve el valor booleano `verdadero` si las condiciones se satisfacen; de lo contrario, devuelve un valor booleano `falso.`

```
bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero; // asume un válido
  bool signo = falso; // asume que no es signo


  // comprueba una cadena vacia
  if (int(str.length()) == 0) válido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1; // comienza a comprobar los dígitos después del signo
  }

  // comprueba que hay al menos un carácter después del signo
  if (signo && int(str.length() == 1)) valido = falso

  // ahora comprueba la cadena, la cual sabemos que tiene al menos un ca-
     rácter que no es un signo
  i = inicio;
```

```
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso; // encuentra un carácter que
    no es digito
    i++; // se mueve al siguiente carácter
  }

  return valido;
}
```

En el código para el método `isvalidInt()`, debe ponerse atención a las condiciones que se están comprobando. Éstas se comentan en el código y consisten en comprobar lo siguiente:

- La cadena no está vacía.
- Un símbolo de signo válido (+ o −) está presente.
- Si un símbolo de signo está presente, al menos un dígito lo sigue.
- Todos los caracteres restantes en la cadena son dígitos.

Sólo si todas estas condiciones se cumplen la función devuelve un valor booleano `verdadero`. Una vez que es devuelto este valor, la cadena puede ser convertida con seguridad en un número entero con la tranquilidad que no resultará ningún valor inesperado que obstaculice el procesamiento de datos posterior. El programa 7.15 usa este método dentro del contexto de un programa completo.

Programa 7.15

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

int main()
{
  bool isvalidInt(string); // prototipo de la función (declaración)
  string valor;
  int numero;

  cout << "Introduzca un número entero: ";
  getline(cin, valor);
```

(Continúa)

(Continuación)

```
  if (!isvalidInt(valor))
   cout << "El número que introdujo no es un número entero válido.";
  else
  {
      numero = atoi(valor.c_str());
    cout << "El número entero que introdujo es " << número;
  }


  return 0;
}

bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero; // asume un número valido
  bool signo = falso; // asume que no es signo


  // comprueba una cadena vacía
  if (int(str.length()) == 0) valido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1; // comienza a comprobar los dígitos después del signo
  }

  // comprueba que hay al menos un carácter después del signo
  if (signo && int(str.length() == 1)) valido = falso

  // ahora comprueba la cadena, la cual sabemos que tiene al menos un
     carácter que no es un signo
  i = inicio;
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso; // encuentra un carácter
    que no es dígito

    i++; // se mueve al siguiente carácter
  }

  return valido;
}
```

Dos muestras de ejecuciones del programa 7.15 produjeron lo siguiente:

```
Introduzca un número entero: 12e45
El numero que introdujo no es un número entero válido.
```

y

```
Introduzca un número entero: -12345
El número entero que introdujo es -12345
```

Como lo ilustra esta salida, el programa determina con éxito que se introdujo un carácter inválido en la primera ejecución.

Una segunda línea de defensa es proporcionar código de procesamiento de errores dentro del contexto del código de manejo de excepciones. Este tipo de código se proporciona por lo general para permitir al usuario corregir un problema como la introducción de datos inválidos al reintroducir un nuevo valor. El medio para proporcionar esto en C++ se conoce como manejo de excepciones y se presenta a continuación.

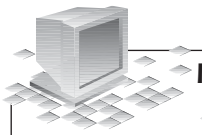
Usando el manejo de excepciones, puede construirse un medio completo para asegurar que el usuario introduce un número entero en respuesta a una solicitud de un valor entero. La técnica que se usará extiende la función `isvalidInt()` incluida en el programa 7.15 para asegurar que un valor entero inválido no sólo es detectado, sino que el programa proporciona al usuario la opción de reintroducir valores hasta que se obtiene un número entero válido. Esta técnica puede aplicarse con facilidad para asegurar la introducción de un número de precisión doble válido, el cual es el otro tipo de datos numéricos solicitado con frecuencia como datos introducidos por el usuario.

Empleando la función `isvalidInt()` proporcionada en el programa 7.15, ahora desarrollamos una función más detallada llamada `getanInt()` que usa el procesamiento de excepciones para aceptar en forma continua una entrada del usuario hasta que se detecta una cadena que corresponde a un número entero válido. Una vez que se ha introducido dicha cadena, la función `getanInt()` convierte la cadena en un número entero y devuelve el valor entero. Esto asegura que el programa que solicita un número entero en realidad recibe un número entero y previene cualesquiera efectos indebidos, como una caída del programa debida a que se introdujo un tipo de datos inválido.

El algoritmo que se usará para llevar a cabo esta tarea es:

```
Establecer una variable booleana llamada notanint como verdadero
while (notanint es verdadero)
    try
        Aceptar un valor de cadena
        Si el valor de cadena no corresponde a un número entero se
        lanza una excepción
    atrapar la excepción
        Desplegar el mensaje de error "Número entero inválido - Por fa-
        vor reintroduzca: "
        Enviar el control de vuelta a la instrucción while
        Establecer notanint en falso (esto causa que termine el ciclo)
  End while
  Devolver el número entero correspondiente a la cadena introducida
```

El código correspondiente a este algoritmo se resalta en el programa 7.16.



Programa 7.16

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

int main()
{
  int getanInt();  // declaración de función (prototipo)
  int valor;

  cout << "Introduzca un valor entero: ";
  valor = getanInt();
  cout << "El número entero introducido es: " << valor << endl;

  return 0;
}
```

```
int getanInt()
{
  bool isvalidInt(string); // declaración de función (prototipo)
  bool notanint = verdadero;
  string segundo_valor;

  while (notanint)
  {
    try
    {
      cin >> segundo_valor; // acepta una entrada de cadena
      if (!isvalidInt(segundo_valor)) throw segundo_valor;
      }
    catch (string e)
    {
      cout << "Número entero inválido - Por favor reintroduzca: ";
        continue; // envia el control a la instruccion while
    }
    notanint = falso;
  }
  return atoi(segundo_valor.c_str()); // convierte en un número entero
}
```

(Continúa)

(Continuación)

```
bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero;  // supone un número válido
  bool signo = falso;  // supone que no es signo

  // comprueba una cadena vacía
  if (int(str.length()) == 0) válido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1;  // inicia la comprobación de digitos después del signo
  }

  // comprueba que haya al menos un carácter después del signo
  if (signo && int(str.length()) == 1) válido = falso;

  // ahora comprueba la cadena, la cual sabemos que al menos tiene un
     carácter que no es un signo
  i = inicio;
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso;  // encuentra un carácter
    que no es un dígito
    i++;  // se mueve al siguiente carácter
  }

 return valido;
}
```

A continuación hay una muestra de la salida producida por el programa 7.16:

```
Introduzca un valor entero: abc
Número entero inválido – Por favor reintroduzca: 12.
Número entero inválido – Por favor reintroduzca: 12e
Número entero inválido – Por favor reintroduzca: 120
El número entero introducido es: 120
```

Como lo muestra esta salida, la función `getanInt()` funciona en forma correcta. Solicita entrada en forma continua hasta que se introduce un número entero válido.

Ejercicios 7.4

1. Escriba un programa en C++ que indique al usuario que escriba un número entero. Haga que su programa acepte el número, como un entero, usando `cin` y, usando `cout,` despliegue el valor que aceptó en realidad su programa de los datos introducidos. Ejecute su programa cuatro veces. La primera vez que lo haga, introduzca un número entero válido, la segunda vez introduzca un número de precisión doble y la tercera vez introduzca un carácter. A continuación, introduzca el valor 12e34.
2. Modifique el programa que escribió para el ejercicio 1 pero haga que su programa use una variable de precisión doble. Ejecute el programa cuatro veces. La primera vez introduzca un número entero, la segunda vez introduzca un número decimal, la tercera vez introduzca un número decimal con una f como el último carácter introducido y la cuarta vez introduzca un carácter. Usando el despliegue de la salida, haga el seguimiento de cuál número aceptó en realidad su programa de los datos que introdujo. ¿Qué sucedió, si es que sucedió algo, y por qué?
3. **a.** ¿Por qué piensa que los programas de aplicación exitosos contienen comprobaciones extensas de la validez de los datos introducidos? (*Sugerencia:* Revise los ejercicios 1 y 2.)
   b. ¿Cuál piensa que sea la diferencia entre la comprobación de un tipo de datos y una comprobación de la racionalidad de los datos?
   c. Suponga que un programa solicita que el usuario introduzca un día, mes y año. ¿Cuáles son algunas comprobaciones razonables que podrían hacerse con los datos introducidos?
4. **a.** Introduzca y ejecute el programa 7.15.
   b. Ejecute el programa 7.15 cuatro veces, usando los datos a los que se hizo referencia en el ejercicio 1 para cada ejecución.
5. Modifique el programa 7.15 para desplegar los caracteres inválidos que se introduzcan.
6. Modifique el programa 7.15 para solicitar un número entero de manera continua hasta que se introduzca un número válido.
7. Modifique el programa 7.15 para eliminar todos los espacios antes y después de la cadena introducida antes que se compruebe su validez.
8. Escriba una función que compruebe cada dígito conforme se introduzca, en lugar de comprobar la cadena completada, como se hace en el programa 7.15.
9. Introduzca y ejecute el programa 7.16.
10. Modifique la función `isvalidInt()` usada en el programa 7.16 para eliminar todos los espacios en blanco antes y después de su argumento de cadena antes de determinar si la cadena corresponde a un número entero válido.
11. Modifique la función `isvalidInt()` usada en el programa 7.16 para aceptar una cadena que termine en un punto decimal. Por ejemplo, introduzca 12. Deberá aceptarse y convertirse en el número entero doce.
12. **a.** Escriba una función de C++ llamada `isvalidReal()` que compruebe un número de precisión doble válido. Esta clase de número puede tener un signo + o – opcional, cuando más un punto decimal, el cual puede ser el primer carácter,

y cuando menos un dígito entre 0 y 9 inclusive. La función deberá devolver un valor booleano de verdadero si el número introducido es un número real; de lo contrario, deberá devolver un valor booleano de falso.

b. Modifique la función `isvalidReal()` escrita para el ejercicio 12a para eliminar todos los espacios en blanco antes y después de su argumento de cadena antes de determinar si la cadena corresponde a un número real válido.

13. Escriba y ejecute una función de C++, llamada `getareal()`, que use el manejo de excepciones para aceptar en forma continua la introducción de una cadena hasta que se introduzca una cadena y pueda convertirse en un número real. La función deberá devolver un valor doble correspondiente al valor de cadena introducido por el usuario.

7.5 Espacio de nombres y creación de una biblioteca personal

Hasta la introducción de las computadoras personales a principios de la década de los años 80, con el uso extenso de circuitos integrados y microprocesadores, tanto la velocidad de las computadoras como su memoria disponible estaban muy restringidas. Por ejemplo, las computadoras más avanzadas de la época tenían velocidades medidas en milisegundos (una milésima de segundo); las computadoras actuales tienen velocidades que se miden en nanosegundos (una mil millonésima de segundo) y superiores. Del mismo modo, la capacidad de memoria de las primeras computadoras de escritorio consistía en 32 mil ubicaciones, con cada ubicación consistente en ocho bits. Las memorias de las computadoras actuales consisten en millones de ubicaciones de memoria, cada una consistente de 32 a 64 bits.

Estas primeras restricciones de hardware hicieron imperativo que los programadores utilizaran todos los trucos posibles para ahorrar espacio de memoria y hacer que los programas se ejecutaran de manera más eficiente. Casi todos los programas se hacían a mano e incluían lo que se conocía como "código inteligente" para minimizar el tiempo de ejecución y maximizar el uso del almacenamiento de memoria. Por desgracia, este código individualizado, con el tiempo, se convirtió en un desastre. Los nuevos programadores tuvieron que dedicar un tiempo considerable para entender el código existente; incluso el programador original tenía problemas para comprender código que había escrito sólo unos meses antes. Esto hacía que las modificaciones consumieran mucho tiempo y fueran muy costosas, e impidió el uso rentable del código existente para nuevas instalaciones.

La incapacidad para reutilizar el código en forma eficiente, combinada con la expansión de las capacidades del hardware, proporcionó el incentivo para instaurar una programación más eficiente. Al principio esto condujo a los conceptos de programación estructurada incorporados en los lenguajes por procedimientos, como Pascal, y en la actualidad a las técnicas orientadas a objetos que forman la base de C++. Sin embargo, una de las primeras críticas a C++ fue que no proporcionaba una biblioteca de clases detallada. Esto ha cambiado en forma impresionante con el estándar ANSI/ISO actual y la inclusión de una biblioteca C++ muy amplia.

No obstante, sin importar cuántas clases y métodos proporcionen, cada tipo de aplicación de programación, como en las áreas de ingeniería, científica y financiera, tienen sus propios requerimientos especializados. Por ejemplo, C++ proporciona funciones de fecha y hora bastante aceptables en su archivo de encabezado `ctime`. Sin embargo, para necesidades especiales, como las que se encuentran en problemas de organización, estas fun-

ciones deben expandirse. Por tanto, un conjunto de funciones más completo incluiría la capacidad de encontrar el número de días hábiles entre dos fechas y que tomara en cuenta fines de semana y días festivos. También requeriría funciones que aplicaran algoritmos del día anterior y el siguiente y tomaran en cuenta los años bisiestos y los días reales en cada mes. Éstos podrían proporcionarse como parte de una clase `Date` más completa o como funciones que no son de clase.

En situaciones como ésta, los programadores crean y comparten sus propias bibliotecas de clases y funciones con otros programadores que trabajan en los mismos proyectos u otros similares. Una vez que se han probado las clases y funciones, pueden incorporarse en cualquier programa sin dedicar más tiempo a la codificación.

En esta etapa en su carrera de programación, puede comenzar a construir su propia biblioteca de funciones y clases especializadas. La sección 7.4 describe cómo puede lograrse esto usando las funciones de validación de entradas, `isvalidInt()` y `getanInt()`, las cuales se reproducen a continuación por comodidad:

```
bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero;  // supone un número valido
  bool signo = falso;  // supone que no es signo

  // comprueba una cadena vacía
  if (int(str.length()) == 0) valido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1; // inicia la comprobación de dígitos después del signo
  }

  // comprueba que haya al menos un carácter después del signo
  if (signo && int(str.length()) == 1) valido = falso;

  // ahora comprueba la cadena, la cual sabemos que al menos tiene un ca-
     rácter que no es un signo
  i = inicio;
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso; // encuentra un carácter que
    no es un dígito
    i++; // se mueve al siguiente carácter
  }

  return valido;
}
int getanInt()
{
  bool isvalidInt(string); // declaración de función (prototipo)
  bool notanint = verdadero;
  string segundo_valor;
```

```
  while (notanint)
  {
    try
    {
      cin >> segundo_valor; // acepta una entrada de cadena
      if (!isvalidInt(segundo_valor)) throw segundo_valor;
    }
    catch (string e)
    {
      cout << "Número entero inválido - Por favor reintroduzca: ";
         continue; // envia el control a la instruccion while
    }
    notanint = falso;
  }
  return atoi(segundo_valor.c_str()); // convierte en un número entero
}
```

El primer paso en la creación de una biblioteca es encapsular todas las funciones y clases preferidas en uno o más espacios de nombres (namespaces) y luego almacenar el código completo (usando o no un espacio de nombres) en uno o más archivos. Por ejemplo, puede crear un espacio de nombres, `comprDatos`, y guardarlo en el archivo llamado `comprDatos.cpp`. Es importante señalar que el nombre de archivo bajo el cual se guarda el espacio de nombres no necesita ser el mismo que el nombre del espacio de nombres usado en el código.

La sintaxis para crear un espacio de nombres es la siguiente:

```
namespace name
{
  aquí van las funciones y/o clases
}  // end of namespace
```

Incluir las dos funciones `isvalidInt()` y `getanInt()` dentro de un espacio de nombres, `comprDatos`, y agregar los archivos `include` apropiados y la instrucción de declaración `using` necesaria para el espacio de nombres nuevo produjo el siguiente código. Por comodidad, la sintaxis requerida para crear el espacio de nombres se ha resaltado:

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

namespace comprDatos
{
  bool isvalidInt(string str)
  {
    int inicio = 0;
    int i;
    bool valido = verdadero;  // supone un número válido
    bool signo = falso;       // supone que no es signo
```

```
    // comprueba una cadena vacía
    if (int(str.length()) == 0) valido = falso;

    // comprueba un signo a la izquierda
    if (str.at(0) == '-'|| str.at(0) == '+')
    {
      signo = verdadero;
      inicio = 1; // inicia la comprobación de dígitos después del signo
    }

    // comprueba que haya al menos un carácter después del signo
    if (signo && int(str.length()) == 1) valido = falso;

    // ahora comprueba la cadena, la cual sabemos que al menos tiene un ca-
       rácter que no es un signo
    i = inicio;
    while(valido && i < int(str.length()))
    {
      if(!isdigit(str.at(i))) valido = falso;  // encuentra un caracter que
      no es un dígito
      i++;  // se mueve al siguiente carácter
    }
    return valido;
  }

  int getanInt()
  {
    bool isvalidInt(string); // declaración de función (prototipo)
    bbool notanint = verdadero;
    string segundo_valor;

    while (notanint)
    {
      try
      {
        cin >> segundo_valor; // acepta una entrada de cadena
        if (!isvalidInt(segundo_valor)) throw segundo_valor;
      }
      catch (string e)
      {
        cout << "Número entero invalido - Por favor reintroduzca: ";
        continue; // envía el control a la instrucción while
      }
      notanint = falso;
    }
    return atoi(segundo_valor.c_str()); // convierte en un número entero
  }
} // fin del namespace comprDatos
```

Una vez que se ha creado el espacio de nombres y se ha almacenado en un archivo, puede incluirse dentro de otro archivo al suministrar una directiva preprocesadora para informar al compilador dónde se encuentra el espacio de nombres deseado y al incluir una directiva `using` que instruya al compilador sobre cuál espacio de nombres particular en el archivo usar. Para el espacio de nombres `comprDatos`, el cual está almacenado en un archivo llamado `comprDatos.cpp`, esto se logra mediante las siguientes instrucciones:

```
#include <c:\\mibiblioteca\\comprDatos>
using namespace comprDatos;
```

La primera instrucción proporciona el nombre de la ruta completa para el archivo del código fuente. Hay que observar que se ha usado un nombre de ruta completo y que se usan dos diagonales inversas para separar los nombres de la ruta. Las diagonales dobles inversas se requieren siempre que se proporciona un nombre de ruta relativo o completo. La única ocasión en que no se requieren las diagonales invertidas es cuando el código de la biblioteca reside en el mismo directorio que el programa que se está ejecutando. Como se indicó, el archivo fuente `comprDatos` está guardado dentro de una carpeta llamada `mibiblioteca`. La segunda instrucción le indica al compilador que use el espacio de nombres `comprDatos` dentro del archivo designado. El programa 7.17 incluye estas dos instrucciones dentro de un programa ejecutable.

Programa 7.17

```
#include <c:\\mibiblioteca\\comprDatos.cpp>
using namespace comprDatos;

int valor;
{
  int value;

  cout << "Introduzca un valor entero: ";
  valor = getanInt();
  cout << "El número entero introducido es: " << valor << endl;



  return 0;
}
```

El único requisito para la instrucción `include` en el programa 7.17 es que el nombre y la ubicación del archivo deben corresponder a un archivo existente que tenga el mismo nombre en la ruta designada; de lo contrario, ocurrirá un error de compilador. Si desea nombrar el archivo del código fuente utilizando una extensión de archivo, puede usarse cualquier extensión en tanto se mantengan las siguientes reglas:

1. El nombre del archivo bajo el cual está almacenado el código incluye la extensión.
2. El mismo nombre de archivo, incluyendo la extensión, se usa en la instrucción `include`.

Por tanto, si el nombre de archivo usado para almacenar las funciones fuera `biblDatos.cpp`, la instrucción `include` en el programa 7.17 sería

```
#include <c:\\mibiblioteca\\biblDatos.cpp>
```

Además, no se requiere un espacio de nombres dentro del archivo. Usar un espacio de nombres nos permite aislar las funciones de comprobación de datos en un área y agregar espacios de nombres adicionales al archivo cuando sea necesario. La designación de un espacio de nombres en la instrucción `using` le indica al compilador que incluya sólo el código en el espacio de nombres especificado, en lugar de todo el código en el archivo. En el programa 7.17, si las funciones de comprobación de datos no estuvieran encerradas dentro de un espacio de nombres, se habría omitido la instrucción `using` para el espacio de nombres `comprDatos`.

Incluir las funciones de comprobación de datos escritas y probadas con anterioridad dentro del programa 7.17 como un archivo separado le permite enfocarse en el código dentro del programa que usa estas funciones, en lugar de estar preocupado con el código de la función en sí. Esto le permite concentrarse en el uso de estas funciones en vez de reexaminar o verificar el código de función escrito y probado antes. En el programa 7.17, el método `main()` ejerce las funciones de comprobación de datos y produce la misma salida que el programa 7.16. Al crear el espacio de nombres `comprDatos`, se ha incluido código fuente para las dos funciones. Esto no se requiere, y en cambio puede guardarse una versión compilada del código fuente. Por último, a un espacio de nombres definido en un archivo pueden hacérseles adiciones en otro archivo usando el mismo nombre del espacio de nombres en el archivo nuevo e incluyendo una instrucción `using` para el espacio de nombres del primer archivo.

Ejercicios 7.5

1. Introduzca y compile el programa 7.17. (*Sugerencia:* El archivo de encabezado de espacio de nombres `comprDatos` y el archivo del programa están disponibles con el código fuente proporcionado en el sitio web de Course Technology para este texto. Véase el ejercicio 4 para el procedimiento de descarga.)

2. ¿Por qué un programador suministraría un archivo de espacio de nombres en su forma compilada y no como código fuente?

3. **a.** ¿Cuál es la ventaja de los espacios de nombres?
   b. ¿Cuál es una posible desventaja de los espacios de nombres?

4. ¿Qué tipos de clases y funciones incluiría en una biblioteca personal? ¿Por qué?

5. **a.** Escriba una función de C++, `entero()`, que devuelve la parte entera de cualquier número transmitido a la función. (*Sugerencia:* Asigne el argumento transmitido a una variable de número entero.)
   b. Incluya la función escrita en el ejercicio 5a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.
   c. Cuando esté seguro que la función `entero()` escrita para el ejercicio 5a funciona en forma correcta, guárdela en un espacio de nombres y en una biblioteca personal de su elección.

6. **a.** Escriba una función de C++, `partefracc()`, que devuelva la parte fraccionaria de cualquier número transmitido a la función. Por ejemplo, si se transmite el número 256.879 a `partefracc()`, deberá devolverse el número .879. Haga que la función `partefracc()` llame a la función `entero()` que escribió en el ejercicio 5a. El número devuelto puede determinarse entonces como el número transmitido a `partefracc()` menos el valor devuelto cuando el mismo argumento es transmitido a `entero()`.
   b. Incluya la función escrita en el ejercicio 6a en un programa que funcione. Asegúrese que la función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.
   c. Cuando esté seguro que la función `partefracc()` escrita para el ejercicio 6a funciona en forma correcta, guárdela en el mismo espacio de nombres y biblioteca personal seleccionados para el ejercicio 5c.

7.6 Errores comunes de programación

Aquí se muestran los errores más comunes asociados con la definición y procesamiento de cadenas:

1. Se ha olvidado incluir el archivo de encabezado `string` cuando se usan objetos de la clase `string`.
2. Se ha olvidado que el carácter de línea nueva, `'\n'`, es un carácter de introducción de datos válido.
3. Se ha olvidado convertir un objeto de clase `string` usando el método `c_str()` cuando se convierten objetos de la clase `string` en tipos de datos numéricos.

7.7 Resumen del capítulo

1. Una cadena literal es cualquier secuencia de caracteres encerrados entre comillas. Una cadena literal se conoce como un valor de cadena, una constante de cadena y, de manera más convencional, una cadena.
2. Una cadena puede construirse como un objeto de la clase `string`.
3. La clase `string` se usa por lo común para construir cadenas con propósitos de entrada y salida, como para indicadores y mensajes desplegados. Debido a sus capacidades, esta clase se usa cuando es necesario comparar o buscar las cadenas o los caracteres individuales en una cadena y necesitan examinarse o extraerse como una subcadena. En situaciones más avanzadas se usa cuando es necesario reemplazar, insertar o eliminar en forma regular los caracteres dentro de una cadena.
4. Las cadenas pueden manipularse usando los métodos de la clase de la que son objetos o usando la cadena de propósito general y métodos de carácter.
5. El objeto `cin`, por sí mismo, tiende a ser de utilidad limitada para la introducción de cadenas debido a que termina la entrada cuando encuentra un espacio en blanco.
6. Para introducir datos a la clase `string` se usa el método `getline()`.
7. El objeto `cout` puede usarse para desplegar las cadenas de la clase `string`.

Consideración de opciones de carrera

Ciencia de materiales e ingeniería metalúrgica

En gran medida, los avances en muchas áreas de la ingeniería en el siglo XX fueron posibles por el descubrimiento de nuevos materiales y una mejor comprensión de las propiedades de los materiales existentes. El conocimiento de los principios físicos y químicos que determinan las propiedades eléctricas de materiales exóticos llamados semiconductores ha producido un progreso fantástico en el campo de los dispositivos de estado sólido, desde transistores hasta chips de circuitos integrados para mainframes. La mejor comprensión de los orígenes de las propiedades metálicas como dureza, fuerza, ductilidad, corrosividad y otras han conducido a una mejora en el diseño de automóviles, aviones, naves espaciales y todo tipo de maquinarias. El campo se subdivide básicamente en metales y no metales, aunque con frecuencia existe una considerable superposición de intereses y actividades.

Ciencia de materiales

La ciencia de materiales se interesa en el comportamiento y propiedades de los materiales, tanto metales como no metales, desde las perspectivas microscópica y macroscópica. Incluye las siguientes áreas:

1. Cerámica. Materiales que no son cristalinos, como el vidrio, que son no metálicos y que requieren altas temperaturas en su procesamiento. La cerámica puede hacerse quebradiza o flexible, dura o suave, o más fuerte que el acero. Puede lograrse que tenga una gran variedad de propiedades químicas.
2. Polímeros. Propiedades estructurales y físicas de polímeros orgánicos, inorgánicos y naturales que son útiles en aplicaciones de ingeniería.
3. Fabricación, procesamiento y tratamiento de materiales. Todos los aspectos de la manufactura de cerámica, metales y síntesis de polímeros, desde el crecimiento de cristales y fibras hasta la formación de metales.
4. Corrosión. Mecanismo de reacción y termodinámica de la corrosión de metales en la atmósfera o sumergidos bajo agua o sustancias químicas, sean fijos o bajo tensión.
5. Tensión-estiramiento y fatiga-fractura de materiales de ingeniería. Las propiedades físicas que rigen la deformación y fractura de materiales y su mejoramiento y uso en la construcción y el diseño.

(continúa)

Consideración de opciones de carrera

Ingeniería metalúrgica

La ingeniería metalúrgica es la rama de la ingeniería responsable de la producción de metales y aleaciones metálicas, desde el descubrimiento de depósitos de menas hasta la fabricación del metal refinado en productos útiles. Los ingenieros metalúrgicos son importantes en todos los pasos de la producción de metales a partir de las menas de metal. La ingeniería metalúrgica incluye las siguientes áreas:

1. Ingeniería de minas. Por lo general una rama separada de la ingeniería. Sin embargo, los intereses de los ingenieros de minas y los metalúrgicos con frecuencia se superponen en los procesos de extracción de metales de las menas de metal y el refinamiento para hacer productos utilizables. La metalurgia de extracción usa las reacciones físicas y químicas para optimizar la producción de metal.
2. Fabricación de metales. Formación de metales en productos como latas, cables y tubos; moldeado y unión de metales; por ejemplo, por soldadura.
3. Metalurgia física. Análisis de las características de tensión-estiramiento, fatiga-fractura de los metales y aleaciones metálicas para prevenir fallas en los componentes de ingeniería.

Parte dos

Programación orientada a objetos

Capítulos