15

Contenedores e iteradores de la Biblioteca estándar

Son los libros,
las artes, las academias,
que muestran, contienen
y nutren a todo el mundo.
—William Shakespeare

Un viaje a través de todo el universo en un mapa.
—Miguel de Cervantes

Objetivos

En este capítulo aprenderá a:

- Ver una introducción a los contenedores, iteradores y algoritmos de la Biblioteca estándar.
- Usar los contenedores de secuencia `vector`, `list` y `deque`.
- Usar los contenedores asociativos `set`, `multiset`, `map` y `multimap`.
- Usar los adaptadores de contenedores `stack`, `queue` y `priority_queue`.
- Usar los iteradores para acceder a los elementos de los contenedores.
- Usar el algoritmo `copy` y los iteradores `ostream_iterator` para imprimir un contenedor.
- Usar el "casi contenedor" `bitset` para implementar la Criba de Eratóstenes y determinar números primos.

Plan general

15.1 Introducción

La Biblioteca estándar define componentes poderosos, basados en plantillas y reutilizables, que implementan muchas estructuras de datos comunes y algoritmos que se utilizan para procesar esas estructuras de datos. Comenzamos a introducir las plantillas en los capítulos 6 y 7, y las utilizamos extensivamente aquí y en los capítulos 16 y 19. En la industria, las características que se presentan en este capítulo se conocen comúnmente como la *Biblioteca de plantillas estándar* o *STL*.[1] Nos referiremos en ocasiones a estas características como la STL. En el documento del estándar de C++, estas características se consideran simplemente como parte de la Biblioteca estándar de C++.

Contenedores, iteradores y algoritmos

En este capítulo presentamos los tres componentes clave de la Biblioteca estándar: **contenedores** (estructuras de datos en *plantillas*), iteradores y algoritmos. Los contenedores son estructuras de datos capaces de almacenar objetos de *casi* cualquier tipo de datos (hay ciertas restricciones). Más adelante veremos que hay tres estilos de clases contenedoras: *contenedores de primera clase*, *adaptadores* y *casi contenedores*.

Funciones miembro comunes entre contenedores

Cada contenedor tiene funciones miembro asociadas; en *todos* los contenedores hay definido un conjunto de estas funciones miembro. Presentamos la mayor parte de esta funcionalidad común en nuestros ejemplos de `array` (que se introdujo en el capítulo 7), `vector` (que se introdujo en el capítulo 7 y que veremos con más detalle aquí), `list` (sección 15.5.2) y `deque` (sección 15.5.3).

Iteradores

Los **iteradores**, que tienen propiedades similares a las de los *apuntadores*, se utilizan para manipular los elementos de los contenedores. Los *arreglos integrados* también se pueden manipular mediante algoritmos de la Biblioteca estándar, utilizando apuntadores como iteradores. Más adelante veremos que es conveniente manipular los contenedores mediante iteradores, ya que nos proporcionan un tremendo poder de expresión al combinarlos con los algoritmos de la Biblioteca estándar; en ciertos casos, se reducen muchas líneas de código a una sola instrucción.

1 La STL fue desarrollada por Alexander Stepanov y Meng Lee en Hewlett-Packard, y se basa en su investigación en el campo de la programación genérica, con contribuciones considerables por parte de David Musser.

Algoritmos
Los **algoritmos** de la Biblioteca estándar son plantillas de funciones que realizan manipulaciones de datos comunes, tales como *búsqueda*, *ordenamiento* y *comparación de elementos o contenedores completos*. La Biblioteca estándar proporciona *muchos* algoritmos. La mayoría utilizan iteradores para acceder a los elementos de un contenedor. Cada algoritmo tiene *requerimientos mínimos* para los tipos de iteradores que se pueden utilizar con éste. Más adelante veremos que los contenedores soportan tipos específicos de iteradores, algunos más poderosos que otros. El tipo de iterador soportado por un *contenedor* determina si éste se puede utilizar con un algoritmo específico. Los iteradores encapsulan los mecanismos utilizados para acceder a los elementos del contenedor. Este encapsulamiento permite aplicar muchos de los algoritmos a varios contenedores, *independientemente* de la implementación subyacente del contenedor. Esto también permite a los programadores crear nuevos algoritmos que puedan procesar los elementos de *múltiples* tipos de contenedores.

Estructuras de datos en plantillas personalizadas
En el capítulo 19 crearemos nuestras *propias* estructuras de datos en plantillas, incluyendo listas enlazadas, colas, pilas y árboles. Entrelazaremos con cuidado los objetos con apuntadores. El código basado en apuntador es complejo y propenso a errores; la más ligera omisión o descuido puede producir graves *violaciones al acceso de memoria* y errores de *fuga de memoria*, sin que el compilador se queje. Si muchos programadores en un proyecto extenso implementan contenedores y algoritmos similares para distintas tareas, el código se vuelve difícil de modificar, mantener y depurar.

Observación de Ingeniería de Software 15.1
Evite reinventar la rueda; programe con los componentes de la Biblioteca estándar de C++.

Tip para prevenir errores 15.1
Los contenedores tipo plantilla pre-empaquetados de la Biblioteca estándar son suficientes para la mayoría de las aplicaciones. El uso de la Biblioteca estándar ayuda a los programadores a reducir el tiempo de prueba y depuración.

Tip de rendimiento 15.1
La Biblioteca estándar se concibió y diseñó teniendo en mente el rendimiento y la flexibilidad.

15.2 Introducción a los contenedores

Los tipos de contenedores de la Biblioteca estándar se muestran en la figura 15.1. Los contenedores se dividen en cuatro categorías principales: **contenedores de secuencia**, **contenedores asociativos ordenados**, **contenedores asociativos desordenados** y **adaptadores de contenedores**.

Clase contenedora	Descripción
Contenedores de secuencia	
`array`	Tamaño fijo. Acceso directo a cualquier elemento.
`deque`	Inserciones y eliminaciones rápidas en la parte inicial o final. Acceso directo a cualquier elemento.
`forward_list`	Lista con enlace simple, inserción y eliminación rápidas en cualquier parte. Nueva en C++11.

Fig. 15.1 | Clases contenedoras de la Biblioteca estándar y adaptadores de contenedores (parte 1 de 2).

Clase contenedora	Descripción
`list`	Lista con enlace doble, inserción y eliminación rápida en cualquier parte.
`vector`	Inserciones y eliminaciones rápidas en la parte final. Acceso directo a cualquier elemento.
Contenedores asociativos ordenados; las claves se mantienen en orden	
`set`	Búsqueda rápida, no se permiten duplicados.
`multiset`	Búsqueda rápida, se permiten duplicados.
`map`	Asignación de uno a uno, no se permiten duplicados, búsqueda rápida basada en claves.
`multimap`	Asignación de uno a varios, se permiten duplicados, búsqueda rápida basada
Contenedores asociativos desordenados	
`unordered_set`	Búsqueda rápida, no se permiten duplicados.
`unordered_multiset`	Búsqueda rápida, se permiten duplicados.
`unordered_map`	Asignación de uno a uno, no se permiten duplicados, búsqueda rápida basada
`unordered_multimap`	Asignación de uno a varios, se permiten duplicados, búsqueda rápida basada
Adaptadores de contenedores	
`stack`	Último en entrar, primero en salir (UEPS).
`queue`	Primero en entrar, primero en salir (PEPS).
`priority_queue`	El elemento de mayor prioridad siempre es el primero en salir.

Fig. 15.1 | Clases contenedoras de la Biblioteca estándar y adaptadores de contenedores (parte 2 de 2).

Generalidades de los contenedores

Los *contenedores de secuencia* representan estructuras de datos *lineales* (es decir, todos sus elementos están conceptualmente "alineados en una fila"), tales como objetos `array`, `vector` y listas enlazadas. En el capítulo 19, Custom Templatized Data Structures, estudiaremos las estructuras de datos enlazadas. Los *contenedores asociativos* son estructuras de datos *no lineales* que por lo general pueden localizar elementos almacenados en ellos rápidamente. Dichos contenedores pueden almacenar conjuntos de valores, o **pares clave/valor**. A partir de C++11, las claves en los contenedores asociativos son *inmutables* (no pueden modificarse). Los contenedores de secuencia y los asociativos se conocen colectivamente como **contenedores de primera clase**. Por lo general, las pilas y las colas son versiones restringidas de los contenedores de secuencia. Por esta razón, la Biblioteca estándar implementa a las plantillas de clases `stack`, `queue` y `priority_queue` como **adaptadores de contenedores** que permiten a un programa ver a un contenedor de secuencia en una manera restringida. La clase `string` soporta la misma funcionalidad que un *contenedor de secuencia*, pero sólo almacena datos tipo carácter.

Casi contenedores

Hay otros tipos de contenedores que se consideran **casi contenedores**: arreglos integrados, objetos `bitset` para mantener conjuntos de valores de banderas y objetos `valarray` para realizar operaciones con *vectores matemáticos* de alta velocidad (que no deben confundirse con el contenedor `vector`). Estos tipos se consideran *casi contenedores* debido a que exhiben algunas de (pero no todas) las capacidades de los *contenedores de primera clase*.

Funciones comunes de los contenedores

La mayoría de los contenedores proporcionan una funcionalidad similar. Muchas operaciones se aplican a todos los contenedores, y otras operaciones se aplican a subconjuntos de contenedores similares.

La figura 15.2 describe las diversas funciones comunes disponibles en la mayoría de los contenedores de la Biblioteca estándar. Los operadores sobrecargados <, <=, >, >=, == y != *no* se proporcionan para contenedores `priority_queue`. Los operadores sobrecargados <, <=, > y >= *no* se proporcionan para los *contenedores asociativos desordenados*. Las funciones miembro `rbegin`, `rend`, `crbegin` y `crend` *no* están disponibles en un contenedor `forward_list`. Antes de usar un contenedor, es conveniente que estudie sus capacidades.

Función miembro	Descripción
constructor predeterminado	Un constructor que *inicializa un contenedor vacío*. Por lo general, cada contenedor cuenta con varios constructores que proporcionan distintas formas de inicializar el contenedor.
constructor de copia	Un constructor que inicializa al contenedor para que sea una *copia de un contenedor* existente del mismo tipo.
constructor de movimiento	Un constructor de movimiento (nuevo en C++11; lo analizamos en el capítulo 24) mueve el contenido de un contenedor existente del mismo tipo a un nuevo contenedor. Esto evita la sobrecarga de copiar cada elemento del contenedor que se pasa como argumento.
destructor	La función destructora para encargarse de la limpieza, una vez que el contenedor ya no es necesario.
`empty`	Devuelve `true` si *no* hay elementos en el contenedor; en caso contrario devuelve `false`.
`insert`	Inserta un elemento en el contenedor.
`size`	Devuelve el número de elementos que hay actualmente en el contenedor.
`operator=` de copia	Copia los elementos de un contenedor a otro.
`operator=` de movimiento	El operador de asignación de movimiento (nuevo en C++11; lo analizamos en el capítulo 24) mueve los elementos de un contenedor a otro. Esto evita la sobrecarga de copiar cada elemento del contenedor que se pasa como argumento.
`operator<`	Devuelve `true` si el primer contenedor es *menor que* el segundo; en caso contrario devuelve `false`.
`operator<=`	Devuelve `true` si el primer contenedor es *menor o igual que* el segundo; en caso contrario devuelve `false`.
`operator>`	Devuelve `true` si el primer contenedor es *mayor que* el segundo; en caso contrario devuelve `false`.
`operator>=`	Devuelve `true` si el primer contenedor es *mayor o igual que* el segundo; en caso contrario devuelve `false`.
`operator==`	Devuelve `true` si el contenido del primer contenedor es *igual que* el segundo; en caso contrario devuelve `false`.
`operator!=`	Devuelve `true` si el contenido del primer contenedor es *distinto que* el segundo; en caso contrario devuelve `false`.
`swap`	Intercambia los elementos de dos contenedores. A partir de C++11, ahora existe una versión de función no miembro de `swap` que intercambia el contenido de sus dos argumentos (que deben ser del mismo tipo de contenedor) mediante el uso de operaciones de movimiento, en vez de operaciones de copia.

Fig. 15.2 | Funciones miembro comunes para la mayoría de los contenedores de la Biblioteca estándar (parte 1 de 2).

Función miembro	Descripción
`max_size`	Devuelve el *número máximo de elementos* para un contenedor.
`begin`	Se sobrecarga para devolver un `iterator` o un `const_iterator` que hace referencia al *primer elemento* del contenedor.
`end`	Se sobrecarga para devolver un `iterator` o un `const_iterator` que hace referencia a la *siguiente posición después del final* del contenedor.
`cbegin` (C++11)	Devuelve un `const_iterator` que hace referencia al *primer elemento* del contenedor.
`cend` (C++11)	Devuelve un `const_iterator` que hace referencia a la *siguiente posición después del final* del contenedor.
`rbegin`	Las dos versiones de esta función devuelven ya sea un `reverse_iterator` o un `const_ reverse_iterator` que hace referencia al *último elemento* del contenedor.
`rend`	Las dos versiones de esta función devuelven ya sea un `reverse_iterator` o un `const_ reverse_iterator` que hace referencia a la *posición que está antes del primer elemento* del contenedor.
`crbegin` (C++11)	Devuelve un `const_reverse_iterator` que hace referencia al *último elemento* del contenedor.
`crend` (C++11)	Devuelve un `const_reverse_iterator` que hace referencia a la *posición anterior al primer elemento* del contenedor.
`erase`	Elimina *uno o más* elementos del contenedor.
`clear`	Elimina *todos* los elementos del contenedor.

Fig. 15.2 | Funciones miembro comunes para la mayoría de los contenedores de la Biblioteca estándar (parte 2 de 2).

Tipos anidados comunes de los contenedores de primera clase

La figura 15.3 muestra los *tipos anidados* comunes de los contenedores de primera clase (tipos definidos dentro de la definición de clase de cada contenedor). Estos elementos se utilizan en declaraciones de variables basadas en plantillas, parámetros a funciones y valores de retorno de las funciones (como veremos en este capítulo y en el 16). Por ejemplo, `value_type` en cada contenedor representa siempre el tipo de elementos almacenados en el contenedor. La clase `forward_list` no proporciona los tipos `reverse_iterator` y `const_reverse_iterator`.

`typedef`	Descripción
`allocator_type`	El tipo del objeto utilizado para asignar la memoria del contenedor; no se incluye en la plantilla de clases `array`.
`value_type`	El tipo de elemento almacenado en el contenedor.
`reference`	Una referencia al tipo de elemento del contenedor.
`const_reference`	Una referencia al tipo de elemento del contenedor que sólo puede utilizarse para *leer* elementos y colocarlos en el contenedor, y para realizar operaciones `const`.
`pointer`	Un apuntador al tipo de elemento del contenedor.

Fig. 15.3 | Tipos anidados que se encuentran en los contenedores de primera clase (parte 1 de 2).

typedef	Descripción
`const_pointer`	Un apuntador para el tipo de elemento del contenedor que puede usarse sólo para *leer* elementos y realizar operaciones `const`.
`iterator`	Un iterador que apunta al tipo de elemento del contenedor.
`const_iterator`	Un iterador que apunta a un elemento del tipo de elemento del contenedor. Se utiliza sólo para *leer* elementos y realizar operaciones `const`.
`reverse_iterator`	Un iterador inverso que apunta al tipo de elemento del contenedor. Se utiliza para iterar a través de un contenedor en sentido inverso.
`const_reverse_iterator`	Un iterador inverso que apunta a un elemento del tipo de elemento del contenedor y que puede utilizarse sólo para *leer* elementos y realizar operaciones `const`. Se usa para iterar a través de un contenedor en sentido inverso.
`difference_type`	El tipo del resultado obtenido al restar dos iteradores que hacen referencia al mismo contenedor (`operator-` no está definido para iteradores de contenedores `list` y contenedores asociativos).
`size_type`	El tipo utilizado para contar elementos en un contenedor e indizar a través de un contenedor de secuencia (no se puede indizar a través de un contenedor `list`).

Fig. 15.3 | Tipos anidados que se encuentran en los contenedores de primera clase (parte 2 de 2).

Requerimientos para los elementos de los contenedores

Antes de utilizar un contenedor de la Biblioteca estándar, es importante asegurar que el tipo de objetos que vayan a almacenarse en el contenedor soporten un conjunto *mínimo* de funcionalidad. Al insertar un objeto en un contenedor, se crea una *copia* de ese elemento. Por esta razón, el tipo de objeto debe proporcionar un *constructor de copia* y un *operador de asignación de copia* (versiones personalizadas o predeterminadas, dependiendo de si la clase usa memoria dinámica). Además, los *contenedores asociativos ordenados* y muchos algoritmos requieren de la *comparación* de elementos. Por esta razón, el tipo del objeto debe proporcionar *operadores menor que (<)* y de *igualdad (==)*. A partir de C++11, los objetos también pueden *moverse* a elementos de contenedores, en cuyo caso el tipo del objeto necesita un *constructor de movimiento* y un *operador de asignación de movimiento*; en el capítulo 24 hablaremos sobre la *semántica de movimiento*.

15.3 Introducción a los iteradores

Los *iteradores* tienen muchas características en común con los *apuntadores* y se utilizan para apuntar a los elementos de los *contenedores de primera clase* y para unos cuantos propósitos más. Los iteradores almacenan información de *estado* susceptible para los contenedores específicos en los que operan; por lo tanto, se implementan iteradores para cada tipo de contenedor. Ciertas operaciones de los iteradores son uniformes para todos los contenedores. Por ejemplo, el *operador de desreferencia (*)* actúa sobre un iterador para que el programador pueda utilizar el elemento al que apunta. La *operación* `++` *en un iterador* lo desplaza al *siguiente elemento* del contenedor (algo muy parecido a incrementar un apuntador en un arreglo integrado para que apunte a su siguiente elemento).

Los *contenedores de primera clase* cuentan con las funciones miembro `begin` y `end`. La función **`begin`** devuelve un iterador que apunta al *primer* elemento del contenedor. La función **`end`** devuelve un iterador que apunta al *primer elemento más allá del final del contenedor* (uno más allá del final): un elemento no

existente que se utiliza con frecuencia para determinar cuándo se llega al final de un contenedor. Si el iterador `i` apunta a un elemento específico, entonces la operación `++i` apunta al "siguiente" elemento y `*i` se refiere al elemento al que apunta `i`. El iterador que resulta de `end` se utiliza comúnmente en una comparación de igualdad o desigualdad, para determinar si el "iterador móvil" (`i` en este caso) ha llegado al final del contenedor.

Un objeto de tipo `iterator` de un contenedor hace referencia al elemento de un contenedor que *puede* modificarse. Un objeto de tipo `const_iterator` de un contenedor hace referencia al elemento de un contenedor que *no puede* modificarse.

Uso de `istream_iterator` para entrada y de `ostream_iterator` para salida

Utilizamos los iteradores con **secuencias** (también conocidas como **rangos**). Estas secuencias pueden estar en contenedores, o pueden ser **secuencias de entrada** o de **salida**. El programa de la figura 15.4 demuestra cómo se introducen datos desde la entrada estándar (una secuencia de datos para introducir en un programa) utilizando un **`istream_iterator`**, y cómo se envían datos a la salida estándar (una secuencia de datos como salida de un programa) utilizando un **`ostream_iterator`**. El programa recibe como entrada dos enteros por parte del usuario desde el teclado y muestra la suma de los mismos. Como veremos posteriormente en este capítulo, se pueden usar iteradores `istream_iterator` y `ostream_iterator` con los algoritmos de la Biblioteca estándar para crear poderosas instrucciones. Por ejemplo, podemos usar un `ostream_iterator` con el algoritmo de copia para copiar *todo* el contenido de un contenedor al flujo de salida estándar con una sola instrucción.

```
1  // Fig. 15.4: fig15_04.cpp
2  // Demostración de las operaciones de entrada y salida con iteradores.
3  #include <iostream>
4  #include <iterator> // ostream_iterator e istream_iterator
5  using namespace std;
6
7  int main()
8  {
9     cout << "Escriba dos enteros: ";
10
11    // crea istream_iterator para leer valores int de cin
12    istream_iterator< int > entradaInt( cin );
13
14    int numero1 = *entradaInt; // lee int de la entrada estándar
15    ++entradaInt; // desplaza el iterador al siguiente valor de entrada
16    int numero2 = *entradaInt; // lee int de la entrada estándar
17
18    // crea ostream_iterator para escribir valores int a cout
19    ostream_iterator< int > salidaInt( cout );
20
21    cout << "La suma es: ";
22    *salidaInt = numero1 + numero2; // envía el resultado a cout
23    cout << endl;
24 } // fin de main
```

```
Escriba dos enteros: 12 25
La suma es: 37
```

Fig. 15.4 | Demostración de operaciones de entrada y salida con iteradores.

`istream_iterator`

En la línea 12 se crea un `istream_iterator` capaz de *extraer* (introducir) valores `int` del objeto de entrada estándar `cin`. En la línea 14 se *desreferencia* el iterador `entradaInt` para leer el primer entero de

`cin` y asigna ese entero a `numero1`. El operador de desreferencia * que se aplica al iterador `entradaInt` obtiene el valor del flujo asociado con `entradaInt`; esto es similar al *desreferenciamiento de un apuntador*. En la línea 15 se coloca el iterador `entradaInt` en el siguiente valor del flujo de entrada. En la línea 16 se introduce el siguiente entero desde `entradaInt` y lo asigna a `numero2`.

`ostream_iterator`

En la línea 19 se crea un `ostream_iterator` que es capaz de insertar (enviar) valores `int` en el objeto de salida estándar `cout`. En la línea 22 se envía un entero a `cout` asignando a `*salidaInt` la suma de `numero1` y `numero2`. Observe que usamos el iterator desreferenciado `salidaInt` como *lvalue* en la instrucción de asignación. Si desea enviar otro valor utilizando `salidaInt`, el iterador debe incrementarse con ++ primero. Puede utilizarse el incremento de prefijo o postfijo; utilizamos la forma de prefijo por cuestiones de *rendimiento*, debido a que no crea un objeto temporal.

Tip para prevenir errores 15.2

*Cuando se aplica el operador * (desreferencia) a un iterador `const`, devuelve una referencia `const` al elemento del contenedor, deshabilitando así el uso de las funciones miembro que no son `const`.*

Categorías de iteradores y la jerarquía de categorías de iteradores

La figura 15.5 muestra las categorías de iteradores. Cada categoría ofrece un conjunto específico de funcionalidad. La figura 15.6 muestra la jerarquía de categorías de iteradores. A medida que vaya siguiendo la jerarquía desde la parte inferior hasta la superior, notará que cada categoría de iteradores soporta toda la funcionalidad de las categorías *debajo* de ella en la figura. Por lo tanto, los tipos de iteradores "más débiles" se encuentran en la parte inferior y el tipo de iterador más poderoso se encuentra en la parte superior. Observe que ésta *no* es una jerarquía de herencia.

Categoría	Descripción
de acceso aleatorio	Combina las capacidades de un *iterador bidireccional* con la habilidad para tener acceso *directo* a cualquier elemento del contenedor (es decir, para saltar hacia delante o hacia atrás un número arbitrario de elementos). También pueden compararse con los operadores relacionales.
bidireccional	Combina las capacidades de un *iterador de avance* con la habilidad para desplazarse en dirección *hacia atrás* (es decir, desde el final del contenedor hasta su inicio). Los iteradores bidireccionales soportan algoritmos de varias pasadas.
de avance	Combina las capacidades de los *iteradores de entrada y de salida*, y retiene su posición en el contenedor (como la información sobre el estado). Dichos iteradores pueden usarse para pasar por una secuencia más de una vez (para los denominados algoritmos multipaso).
salida	Se utiliza para escribir un elemento en un contenedor. Un iterador de salida puede desplazarse sólo *hacia delante*, un elemento a la vez. Los iteradores de salida soportan *sólo* algoritmos de una pasada: el mismo iterador de salida *no puede* utilizarse para pasar a través de una secuencia dos veces.
entrada	Utilizada para leer un elemento de un contenedor. Un iterador de entrada puede desplazarse sólo *hacia delante* (es decir, desde el inicio del contenedor hasta su final), un elemento a la vez. Los iteradores de entrada soportan *sólo* algoritmos de una pasada: *no puede* utilizarse el mismo iterador de entrada para pasar a través de una secuencia dos veces.

Fig. 15.5 | Categorías de iteradores.

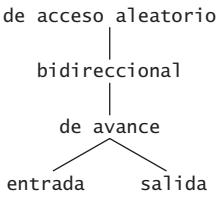

Fig. 15.6 | Jerarquía de categorías de iteradores.

Soporte de los contenedores para los iteradores

La categoría de iteradores que soporta cada contenedor determina si éste puede utilizarse con ciertos algoritmos específicos. *Los contenedores que soportan a los iteradores de acceso aleatorio pueden utilizarse con todos los algoritmos de la Biblioteca estándar*, con la excepción de que si un algoritmo requiere cambios en el tamaño de un contenedor, no podrá usarse en arreglos integrados o en objetos `array`. Los apuntadores a los arreglos *integrados* pueden utilizarse en vez de los iteradores en la mayoría de los algoritmos. La figura 15.7 muestra la categoría de iteradores de cada uno de los contenedores. Los contenedores de primera clase, los objetos `string` y los arreglos integrados pueden recorrerse con iteradores.

Contenedor	Tipo de iterador	Contenedor	Tipo de iterador
Contenedores de secuencia (primera clase)		*Contenedores asociativos desordenados (primera clase)*	
`vector`	acceso aleatorio	`unordered_set`	bidireccional
`array`	acceso aleatorio	`unordered_multiset`	bidireccional
`deque`	acceso aleatorio	`unordered_map`	bidireccional
`list`	bidireccional	`unordered_multimap`	bidireccional
`forward_list`	de avance		
Contenedores asociativos ordenados (primera clase)		*Adaptadores de contenedores*	
`set`	bidireccional	`stack`	ninguno
`multiset`	bidireccional	`queue`	ninguno
`map`	bidireccional	`priority_queue`	ninguno
`multimap`	bidireccional		

Fig. 15.7 | Tipos de iteradores soportados por cada contenedor.

Elementos `typedef` *de iteradores predefinidos*

La figura 15.8 muestra a los elementos `typedef` de los iteradores predefinidos que se encuentran en las definiciones de clase de los contenedores de la Biblioteca estándar. No todos los `typedef` están definidos para cada contenedor. Nosotros utilizamos versiones `const` de los iteradores para recorrer *contenedores* `const` o contenedores que no son `const` y no deben modificarse. Utilizamos *iteradores inversos* para recorrer los contenedores en dirección *inversa*.

Tip para prevenir errores 15.3

Las operaciones realizadas sobre un `const_iterator` *devuelven referencias* `const` *para evitar que se modifiquen los elementos del contenedor que se está manipulando. Utilizar iteradores del tipo* `const_iterator` *siempre que sea apropiado es otro ejemplo del principio del menor privilegio.*

Elementos `typedef` predefinidos para los tipos de iteradores	Dirección de ++	Capacidad
`iterator`	avance	leer/escribir
`const_iterator`	avance	leer
`reverse_iterator`	retroceso	leer/escribir
`const_reverse_iterator`	retroceso	leer

Fig. 15.8 | Elementos `typedef` de los iteradores.

Operaciones con iteradores

La figura 15.9 muestra operaciones que pueden realizarse con cada tipo de iterador. Además de los operadores que se muestran para todos los iteradores, éstos deben proveer constructores predeterminados, constructores de copia y operadores de asignación de copia. Un iterador *de avance* soporta ++ y todas las capacidades de los iteratores de *entrada* y *salida*. Un iterador *bidireccional* soporta -- y todas las capacidades de los iteradores *de avance*. Un iterador de *acceso aleatorio* soporta todas las operaciones mostradas en la tabla. Para los iteradores de entrada y los de salida, no es posible guardar el iterador y utilizar después el valor guardado.

Operación de iterador	Descripción
Todos los iteradores	
`++p`	Preincrementar un iterador.
`p++`	Postincrementar un iterador.
`p = p1`	Asignar un iterador a otro.
Iteradores de entrada	
`*p`	Desreferenciar un iterador como un *rvalue*.
`p->m`	Usar el iterador para leer el elemento `m`.
`p == p1`	Comparar igualdad de iteradores.
`p != p1`	Comparar desigualdad de iteradores.
Iteradores de salida	
`*p`	Desreferenciar un iterador como un *lvalue*.
`p = p1`	Asignar un iterador a otro.
Iteradores de avance	Los iteradores de avance proporcionan toda la funcionalidad de los iteradores de entrada y los de salida.
Iteradores bidireccionales	
`--p`	Predecrementar un iterador.
`p--`	Postdecrementar un iterador.
Iteradores de acceso aleatorio	
`p += i`	Incrementar el iterador `p` en `i` posiciones.
`p -= i`	Decrementar el iterador `p` en `i` posiciones.

Fig. 15.9 | Operaciones con iteradores para cada tipo de iterador (parte 1 de 2).

Operación de iterador	Descripción
`p + i` *o* `i + p`	El valor de la expresión es un iterador posicionado en `p`, incrementado en `i` posiciones.
`p - i`	El valor de la expresión es un iterador posicionado en `p`, decrementado en `i` posiciones.
`p - p1`	El valor de la expresión es un entero que representa la distancia entre dos elementos en el mismo contenedor.
`p[ i ]`	Devuelve una referencia al desplazamiento del elemento de `p`, por `i` posiciones.
`p < p1`	Devuelve `true` si el iterador `p` es *menor que* el iterador `p1` (es decir, si el iterador `p` se encuentra *antes* que el iterador `p1` en el contenedor); en cualquier otro caso devuelve `false`.
`p <= p1`	Devuelve `true` si el iterador `p` es *menor o igual* que el iterador `p1` (es decir, si el iterador `p` se encuentra *antes* o *en la misma ubicación* que el iterador `p1` en el contenedor); en cualquier otro caso devuelve `false`.
`p > p1`	Devuelve `true` si el iterador `p` es *mayor que* el iterador `p1` (es decir, si el iterador `p` se encuentra *después* que el iterador `p1` en el contenedor); en cualquier otro caso devuelve `false`.
`p >= p1`	Devuelve `true` si el iterador `p` es *mayor o igual que* el iterador `p1` (es decir, si el iterador `p` se encuentra *después* o *en la misma ubicación* que el iterador `p1` en el contenedor); en cualquier otro caso devuelve `false`.

Fig. 15.9 | Operaciones con iteradores para cada tipo de iterador (parte 2 de 2).

15.4 Introducción a los algoritmos

La Biblioteca estándar provee muchos de los *algoritmos* que usted utilizará frecuentemente para manipular una variedad de contenedores. Los algoritmos para *insertar*, *eliminar*, *buscar*, *ordenar* y demás son apropiados para algunos o todos los contenedores de secuencia y asociativos. *Los algoritmos operan sobre los elementos de los contenedores solamente en forma indirecta, a través de los iteradores*. Muchos algoritmos operan en secuencias de elementos definidas por iteradores que apuntan al *primer elemento* de la secuencia y a *un elemento más allá del último elemento*. Además, es posible *crear sus propios nuevos algoritmos* que operen en una forma similar, de manera que puedan utilizarse con los contenedores e iteradores de la Biblioteca estándar. En este capítulo usaremos el algoritmo `copy` en muchos ejemplos para copiar el contenido de un contenedor en la entrada estándar. En el capítulo 16 veremos muchos algoritmos de la Biblioteca estándar.

15.5 Contenedores de secuencia

La Biblioteca de plantillas estándar de C++ cuenta con cinco *contenedores de secuencia*: `array`, `vector`, `deque`, `list` y `forward_list`. Las plantillas de clases `array`, `vector` y `deque` se basan en arreglos integrados. Las plantillas de clases `list` y `forward_list` implementan estructuras de datos de listas enlazadas, que veremos en el capítulo 19. Ya hemos visto y utilizado la plantilla de clase `array` extensivamente, por lo que no la cubriremos aquí de nuevo.

Rendimiento y elección del contenedor apropiado

En la figura 15.2 presentamos las operaciones comunes para la *mayoría* de los contenedores de la Biblioteca estándar. Además de estas operaciones, cada contenedor proporciona generalmente una variedad de capacidades adicionales. Muchas de estas son comunes para varios contenedores, pero no siempre son igual de eficientes para cada contenedor.

Observación de Ingeniería de Software 15.2

Por lo general es preferible reutilizar los contenedores de la Biblioteca estándar en vez de desarrollar estructuras de datos tipo plantillas personalizadas. Para los novatos, `vector` *satisfface comúnmente la mayoría de las aplicaciones.*

Tip de rendimiento 15.2

La inserción en la parte final de un `vector` *es un proceso eficiente. El* `vector` *simplemente crece, si es necesario, para dar cabida al nuevo elemento. Es costoso insertar (o eliminar) un elemento a la mitad de un* `vector`*; toda la porción completa del* `vector` después *del punto de inserción (o eliminación) debe desplazarse, ya que los elementos de un* `vector` *ocupan celdas contiguas en memoria.*

Tip de rendimiento 15.3

Las aplicaciones que requieren de inserciones y eliminaciones frecuentes en ambos extremos de un contenedor por lo general utilizan un contenedor `deque`*, en vez de un* `vector`*. Aunque podemos insertar y eliminar elementos en la parte inicial y final tanto de un* `vector` *como de un* `deque`*, la clase* `deque` *es más eficiente que* `vector` *para llevar a cabo inserciones y eliminaciones en la parte inicial.*

Tip de rendimiento 15.4

Las aplicaciones con inserciones y eliminaciones frecuentes a la mitad y/o a los extremos de un contenedor por lo general utilizan un contenedor `list`*, debido a su eficiente implementación de los procesos de inserción y eliminación en cualquier parte de la estructura de datos.*

15.5.1 Contenedor de secuencia `vector`

La plantilla de clase `vector`, que presentamos en la sección 7.10, ofrece una estructura de datos con ubicaciones *contiguas* en memoria. Esto permite un acceso eficiente y directo a cualquier elemento de un `vector` mediante el operador de subíndice `[]`, en forma idéntica a un arreglo integrado. Al igual que la plantilla de clase `array`, la plantilla `vector` se utiliza más comúnmente cuando los datos en el contenedor deben ser fácilmente accesibles mediante un subíndice, o deben ordenarse, y cuando el número de elementos tal vez necesite aumentar. Cuando la memoria de un `vector` se agota, éste *asigna* un arreglo integrado más extenso, *copia* (o *mueve*; capítulo 24) los elementos originales en el nuevo arreglo integrado y *desasigna* el arreglo integrado anterior.

Tip de rendimiento 15.5

Seleccione el contenedor `vector` *con el mejor rendimiento de acceso aleatorio en un contenedor que pueda crecer.*

Tip de rendimiento 15.6

Los objetos de la plantilla de clase `vector` *ofrecen un acceso indizado rápido mediante el operador de subíndice* `[]` *sobrecargado, ya que se almacenan en* memoria contigua *al igual que un arreglo integrado o un objeto* `array`*.*

Uso de objetos `vector` *y de los iteradores*

La figura 15.10 muestra varias funciones de la plantilla de clase `vector`. Muchas de estas funciones están disponibles en todos los *contenedores de primera clase*. Debe incluir el encabezado `<vector>` para utilizar la plantilla de clase `vector`.

```
// Fig. 15.10: fig15_10.cpp
// La plantilla de clase vector de la Biblioteca estándar.
#include <iostream>
#include <vector> // definición de la plantilla de clase vector
using namespace std;

// prototipo para la plantilla de función imprimirVector
template < typename T > void imprimirVector( const vector< T > &enteros2 );

int main()
{
   const size_t TAMANIO = 6; // define el tamaño del arreglo
   int valores[ TAMANIO ] = { 1, 2, 3, 4, 5, 6 }; // inicializa los valores
   vector< int > enteros; // crea un vector de valores int

   cout << "El tamanio inicial de enteros es: " << enteros.size()
      << "\nLa capacidad inicial de enteros es: " << enteros.capacity() ;

   // la función push_back está en vector, deque y list
   enteros.push_back( 2 );
   enteros.push_back( 3 );
   enteros.push_back( 4 );

   cout << "\nEl tamanio de enteros es: " << enteros.size()
      << "\nLa capacidad de enteros es: " << enteros.capacity() ;
   cout << "\n\nImpresion de un arreglo usando notacion de apuntador: ";

   // muestra el arreglo usando notación de apuntador
   for ( const int *ptr = begin( valores ); ptr != end( valores ); ++ptr )
      cout << *ptr << ' ';

   cout << "\nImpresion de vector usando notacion de iterador: ";
   imprimirVector( enteros );
   cout << "\nContenido invertido del vector enteros: ";

   // muestra el vector en orden inverso usando const_reverse_iterator
   for ( auto iteradorInverso = enteros.crbegin();
      iteradorInverso!= enteros.crend(); ++iteradorInverso )
      cout << *iteradorInverso << ' ';

   cout << endl;
} // fin de main

// plantilla de función para mostrar los elementos de un vector
template < typename T > void imprimirVector( const vector< T > &enteros2 )
{
   // muestra los elementos del vector usando const_iterator
   for ( auto iteradorConst = enteros2.cbegin();
      iteradorConst != enteros2.cend(); ++iteradorConst )
      cout << *iteradorConst << ' ';
} // fin de la función imprimirVector
```

Fig. 15.10 | Plantilla de clase `vector` de la Biblioteca estándar (parte 1 de 2).

```
El tamanio inicial de enteros es: 0
La capacidad inicial de enteros es: 0
El tamanio de enteros es: 3
La capacidad de enteros es: 4

Impresion de un arreglo usando notacion de apuntador: 1 2 3 4 5 6
Impresion de vector usando notacion de iterador: 2 3 4
Contenido invertido del vector enteros: 4 3 2
```

Fig. 15.10 | Plantilla de clase `vector` de la Biblioteca estándar (parte 2 de 2).

Creación de un `vector`

La línea 14 define una instancia llamada `enteros` de la plantilla de clase `vector` que almacena valores `int`. Al crear la instancia de este objeto, se crea un `vector` vacío con tamaño 0 (es decir, el número de elementos almacenados en el `vector`) y capacidad 0 (es decir, el número de elementos que pueden almacenarse sin asignar más memoria al `vector`).

Funciones miembro `size` *y* `capacity` *de* `vector`

Las líneas 16 y 17 demuestran el uso de las funciones `size` y `capacity`; cada una devuelve inicialmente 0 para el `vector` `enteros` en este ejemplo. La función `size` (disponible en *todos* los contenedores excepto `forward_list`) devuelve el número de elementos actualmente almacenados en el contenedor. La función `capacity` (específica para `vector` y `deque`) devuelve el número de elementos que pueden almacenarse en el `vector` antes de que éste *cambie de tamaño dinámicamente* para dar cabida a más elementos.

Función miembro `push_back` *de* `vector`

Las líneas de la 20 a la 22 utilizan la función `push_back` (disponible en todos los *contenedores de secuencia* excepto `array` y `forward_list`) para agregar un elemento al final del `vector`. Si se agrega un elemento a un `vector` lleno, éste aumenta su tamaño; algunas implementaciones hacen que el `vector` aumente su tamaño al *doble*. Los contenedores de secuencia distintos de `array` y `vector` también cuentan con una función `push_front`.

Tip de rendimiento 15.7

El proceso de duplicar el tamaño de un `vector` cuando se necesita más espacio puede provocar un desperdicio considerable del mismo. Por ejemplo, un `vector` lleno con 1 000 000 de elementos cambia su tamaño para dar cabida a 2 000 000 de elementos cuando se agrega uno nuevo. Esto deja a 999 999 elementos sin usar. Los programadores pueden utilizar a `resize` y `reserve` para controlar mejor el uso del espacio.

Actualización de `size` *y* `capacity` *después de modificar un* `vector`

Las líneas 24 y 25 utilizan `size` y `capacity` para ilustrar el nuevo tamaño y capacidad del `vector` después de las tres operaciones con `push_back`. La función `size` devuelve 3: el número de elementos agregados al `vector`. La función `capacity` devuelve 4 (aunque esto podría variar según el compilador), indicando que podemos agregar un elemento adicional sin necesidad de asignar más memoria para el `vector`. Cuando agregamos el primer elemento, el `vector` asignó espacio para un elemento y el tamaño se hizo 1 para indicar que el `vector` sólo contenía un elemento. Cuando agregamos el segundo elemento, la capacidad se *duplicó* a 2 y el tamaño también se hizo 2. Cuando agregamos el tercer elemento, la capacidad se volvió a duplicar para quedar en 4. Por lo tanto, podemos agregar otro elemento antes de que

el `vector` necesite asignar más espacio. Cuando el `vector` se llene en un momento dado y el programa intente agregar un elemento más, duplicará su capacidad a 8 elementos.

Crecimiento de un `vector`

La forma en que crece un `vector` para dar cabida a más elementos (una operación que consume mucho tiempo) *no* está especificada por el Documento del estándar de C++. Los implementadores de la biblioteca de C++ utilizan varios esquemas astutos para minimizar la sobrecarga de *cambiar el tamaño* de un `vector`. Por ende, la salida de este programa puede variar, dependiendo de la versión de `vector` que incluya su compilador. Algunos implementadores de bibliotecas asignan una capacidad inicial extensa. Si un `vector` almacena un pequeño número de elementos, dicha capacidad puede ser un desperdicio de espacio. Sin embargo, puede mejorar considerablemente el rendimiento si un programa agrega muchos elementos a un `vector` y no tiene que reasignar memoria para dar cabida a esos elementos. Esta es una clásica *concesión entre espacio y tiempo*. Los implementadores de bibliotecas deben balancear la cantidad de memoria utilizada y la cantidad de tiempo requerido para realizar varias operaciones con objetos `vector`.

Imprimir el contenido de arreglos integrados con apuntadores

Las líneas 29 y 30 demostraron cómo mostrar el contenido del arreglo integrado `valores` mediante el uso de apuntadores y aritmética de apuntadores. Los apuntadores a un arreglo integrado pueden usarse como iteradores. Si recuerda, en la sección 8.5 vimos que las funciones `begin` y `end` de C++11 (línea 29) del encabezado `<iterator>` reciben cada una un arreglo integrado como argumento. La función `begin` devuelve un iterador que apunta al primer elemento del arreglo integrado, y la función `end` devuelve un iterador que representa la posición que está un elemento *después* del final del arreglo integrado. Las funciones `begin` y `end` *también* pueden recibir objetos contenedores como argumentos. Observe que utilizamos el operador `!=` en la condición de continuación de ciclo. Al iterar usando apuntadores a elementos de arreglos integrados, es común que la condición de continuación de ciclo evalúe si el apuntador ha llegado al final del arreglo integrado. Esta técnica la utilizan con frecuencia los algoritmos de la biblioteca estándar.

Imprimir el contenido de un `vector` *con iteradores*

La línea 33 llama a la función `imprimirVector` (definida en las líneas 45 a 51) para mostrar el contenido de un `vector`, utilizando iteradores. La función recibe una referencia a un `vector const`. La instrucción `for` en las líneas 48 a 50 inicializa la variable de control `iteradorConst` mediante el uso de la función miembro **`cbegin`** de `vector` (nueva en C++11), que devuelve un `const_iterator` al primer elemento en el `vector`. Inferimos el tipo de la variable de control (`vector<int>::const_iterator`) mediante el uso de la palabra clave `auto`. Antes de C++11 se utilizaba la función miembro sobrecargada `begin` para obtener el `const_iterator` (cuando se llama en un contenedor `const`, `begin` devuelve un `const_iterator`. La otra versión de `begin` devuelve un `iterator` que puede usarse para contenedores que no son `const`.

El ciclo continúa mientras que `iteradorConst` no llegue al final del `vector`. Esto se determina mediante la comparación de `iteradorConst` con el resultado de llamar a la función miembro **`cend`** de `vector` (también nueva en C++11), la cual devuelve un `const_iterator` que indica la *posición que está después del último elemento* del `vector`. Si `iteradorConst` es igual a este valor, se ha llegado al final del `vector`. Antes de C++11 se usaba la función miembro sobrecargada `end` para obtener el `const_iterator`. Las funciones `cbegin`, `begin`, `cend` y `end` están disponibles para todos los contenedores de primera clase.

El cuerpo del ciclo desreferencia al iterador `iteradorConst` para obtener el valor del elemento actual. Recuerde que el iterador actúa como un apuntador al elemento y que el operador `*` está sobrecargado para devolver una referencia al elemento. La expresión `++iteradorConst` (línea 49) coloca al

iterador en el siguiente elemento del `vector`. Observe que las líneas 48 a 50 podrían haberse reemplazado con la siguiente instrucción `for` basada en rango:

```
for ( auto const &elemento : enteros2 )
   cout << elemento << ' ';
```

Error común de programación 15.1

Tratar de desreferenciar a un iterador colocado fuera de su contenedor es un error en tiempo de ejecución. En específico, el iterador devuelto por `end` no debe desreferenciarse ni incrementarse.

Mostrar el contenido del `vector` *en sentido inverso con iteradores* `const_reverse_iterator`

En las líneas 37 a 39 se utiliza una estructura `for` (similar a la de la función `imprimirVector`) para iterar a través del `vector` en sentido inverso. Ahora, C++11 incluye las funciones miembro **`crbegin`** y **`crend`** de `vector`, las cuales devuelven iteradores `const_reverse_iterator` que representan los puntos inicial y final al iterar a través de un contenedor en sentido inverso. La mayoría de los contenedores de primera clase soportan este tipo de iterador. Al igual que con las funciones `cbegin` y `cend`, antes de C++11 se utilizaban las funciones miembro sobrecargadas **`rbegin`** y **`rend`** para obtener iteradores `const_reverse_iterator` o `reverse_iterator`, dependiendo de si el contenedor es o no `const`.

C++11: `shrink_to_fit`

A partir de C++11, podemos pedir a un `vector` o `deque` que devuelva la memoria que no necesita al sistema, mediante una llamada a la función miembro **`shrink_to_fit`**. Esta función *solicita* al contenedor que reduzca su capacidad al número de elementos en el contenedor. De acuerdo con el estándar de C++, las implementaciones pueden *ignorar* esta solicitud, de modo que puedan realizar optimizaciones específicas de la implementación.

Funciones de manipulación de elementos de un `vector`

La figura 15.11 muestra a las funciones que permiten la recuperación y manipulación de los elementos de un `vector`. La línea 16 utiliza un constructor sobrecargado de `vector` que toma dos iteradores como argumentos para inicializar a `enteros`. En la línea 16 se inicializa `enteros` con el contenido del arreglo `valores`, desde el inicio de `valores` hasta (pero sin incluir) `valores.cend()` (que apunta al elemento *después* del final de `valores`). En C++11 podemos usar inicializadores de listas para inicializar objetos `vector`, como en:

```
vector< int > enteros{ 1, 2, 3, 4, 5, 6 };
```

o

```
vector< int > enteros = { 1, 2, 3, 4, 5, 6 };
```

Sin embargo, éstos no cuentan todavía con soporte completo en todos los compiladores. Por esta razón, los ejemplos del capítulo inicializan con frecuencia otros contenedores con contenedores `array`, como en la línea 16.

```
1  // Fig. 15.11: fig15_11.cpp
2  // Prueba de las funciones de manipulación de elementos de la
3  // plantilla de clase vector de la Biblioteca estándar.
4  #include <iostream>
```

Fig. 15.11 | Funciones de manipulación de elementos de la plantilla de clase `vector` (parte 1 de 3).

```
5  #include <array> // definición de la plantilla de clase array
6  #include <vector> // definición de la plantilla de clase vector
7  #include <algorithm> // algoritmo de copia
8  #include <iterator> // iterador ostream_iterator
9  #include <stdexcept> // excepción out_of_range
10 using namespace std;
11
12 int main()
13 {
14    const size_t TAMANIO = 6;
15    array< int, TAMANIO > valores = { 1, 2, 3, 4, 5, 6 };
16    vector< int > enteros( valores.cbegin(), valores.cend() );
17    ostream_iterator< int > salida( cout, " " );
18
19    cout << "El vector enteros contiene: ";
20    copy( enteros.cbegin(), enteros.cend(), salida );
21
22    cout << "\nPrimer elemento de enteros: " << enteros.front()
23       << "\nUltimo elemento de enteros: " << enteros.back();
24
25    enteros[ 0 ] = 7; // establece el primer elemento a 7
26    enteros.at( 2 ) = 10; // establece el elemento en la posición 2 a 10
27
28    // inserta 22 como 2do elemento
29    enteros.insert( enteros.cbegin() + 1, 22 );
30
31    cout << "\n\nContenido del vector enteros despues de los cambios: ";
32    copy( enteros.cbegin(), enteros.cend(), salida );
33
34    // acceso a un elemento fuera de rango
35    try
36    {
37       enteros.at( 100 ) = 777;
38    } // fin de try
39    catch ( out_of_range &fueraDeRango ) // excepción out_of_range
40    {
41       cout << "\n\nExcepcion: "  << fueraDeRango.what();
42    } // fin de catch
43
44    // borra el primer elemento
45    enteros.erase( enteros.cbegin() );
46    cout << "\n\nVector enteros despues de borrar el primer elemento: ";
47    copy( enteros.cbegin(), enteros.cend(), salida );
48
49    // borra los elementos restantes
50    enteros.erase( enteros.cbegin(), enteros.cend() );
51    cout << "\nDespues de borrar todos los elementos, el vector enteros "
52       << ( enteros.empty() ? "esta" : "no esta" ) << " vacio";
53
54    // inserta los elementos del arreglo valores
55    enteros.insert( enteros.cbegin(), valores.cbegin(), valores.cend() );
56    cout << "\n\nContenido del vector enteros antes de clear: ";
57    copy( enteros.cbegin(), enteros.cend(), salida );
```

Fig. 15.11 | Funciones de manipulación de elementos de la plantilla de clase `vector` (parte 2 de 3).

```
58
59     // vacia enteros; clear llama a erase para vaciar una colección
60     enteros.clear();
61     cout << "\nDespues de clear, el vector enteros "
62        << ( enteros.empty() ? "esta" : "no esta" ) << " vacio" << endl;
63  } // fin de main
```

```
El vector enteros contiene: 1 2 3 4 5 6
Primer elemento de enteros: 1
Ultimo elemento de enteros: 6

Contenido del vector enteros despues de los cambios: 7 22 2 10 4 5 6

Excepcion: invalid vector<T> subscript

Vector enteros despues de borrar el primer elemento: 22 2 10 4 5 6
Despues de borrar todos los elementos, el vector enteros esta vacio

Contenido del vector enteros antes de clear: 1 2 3 4 5 6
Despues de clear, el vector enteros esta vacio
```

Fig. 15.11 | Funciones de manipulación de elementos de la plantilla de clase `vector` (parte 3 de 3).

`ostream_iterator`

La línea 17 define un `ostream_iterator` llamado `salida` que puede utilizarse para mostrar los enteros separados por espacios sencillos mediante `cout`. Un `ostream_iterator<int>` muestra sólo valores de tipo `int` o de un tipo compatible. El primer argumento para el constructor especifica el flujo de salida y el segundo argumento es una cadena que especifica el separador para mostrar los valores; en este caso, la cadena contiene un carácter de espacio. Utilizamos el `ostream_iterator` (definido en el encabezado `<iterator>`) para mostrar el contenido del `vector` en este ejemplo.

Algoritmo `copy`

La línea 20 utiliza el algoritmo **`copy`** de la Biblioteca estándar (del encabezado **`<algorithm>`**) para mostrar todo el contenido completo del `vector` `enteros` en la salida estándar. El algoritmo copia cada elemento en un rango, empezando con la posición especificada por el iterador en su primer argumento y hasta (pero *sin* incluir) la posición especificada por el iterador en su segundo argumento. Estos dos argumentos deben satisfacer los requerimientos del *iterador de entrada*: deben ser iteradores a través de los cuales puedan leerse valores de un contenedor, como `const_iterator`. También deben representar un rango de elementos; al aplicar el operador ++ al primer iterador se debe ocasionar eventualmente que éste llegue al segundo argumento iterador en el rango. Los elementos se copian en la ubicación especificada por el *iterador de salida* (es decir, un iterador a través del cual se puede almacenar o mostrar un valor) que se especifica como el último argumento. En este caso, el *iterador de salida* es un `ostream_iterator` que se anexa a `cout`, de manera que los elementos se copian a la salida estándar.

Funciones miembro `front` *y* `back` *de* `vector`

Las líneas 22 y 23 utilizan las funciones **`front`** y **`back`** (disponibles para la mayoría de los *contenedores de secuencia*) para determinar el primer y último elementos del `vector`, respectivamente. Observe la diferencia entre las funciones `front` y `begin`. La función `front` devuelve una referencia al primer elemento en el `vector`, mientras que la función `begin` devuelve un *iterador de acceso aleatorio* que apunta al primer elemento en el `vector`. Observe además la diferencia entre las funciones `back` y `end`. La función `back`

devuelve una referencia al último elemento en el `vector`, mientras que la función `end` devuelve un *iterador de acceso aleatorio* que apunta a la ubicación que está *después* del último elemento.

Error común de programación 15.2

El `vector` no debe estar vacío; de no ser así, los resultados de las funciones `front` y `back` están indefinidos.

Acceso a los elementos de un `vector`

Las líneas 25 y 26 muestran dos maneras de acceder a los elementos de un `vector`. Esto también puede utilizarse con los contenedores `deque`. La línea 25 utiliza el operador de subíndice que se sobrecarga para devolver ya sea una referencia al valor en la ubicación especificada, o una referencia a ese valor `const`, dependiendo de si el contenedor es `const` o no. La función `at` (línea 26) realiza la misma operación, pero con *comprobación de límites*. La función `at` primero comprueba el valor suministrado como argumento y determina si se encuentra dentro de los límites del `vector`. De no ser así, la función `at` lanza una excepción `out_of_range` (como se demuestra en las líneas 35 a 42). La figura 15.12 muestra algunos de los tipos de excepciones de la Biblioteca estándar (los tipos de excepciones de la Biblioteca estándar se describen en el capítulo 17).

Tipo de excepción	Descripción
`out_of_range`	Indica cuando el subíndice está fuera de rango; por ejemplo, cuando se especifica un subíndice inválido en la función miembro at de `vector`.
`invalid_argument`	Indica que se pasó un argumento inválido a una función.
`length_error`	Indica un intento de crear un contenedor demasiado largo, o un objeto `string`, etcétera.
`bad_alloc`	Indica que un intento de asignar memoria con `new` (o con un asignador) falló debido a que no había suficiente memoria disponible.

Fig. 15.12 | Algunos tipos de excepciones en el encabezado `<stdexcept>`.

Función miembro `insert` *de* `vector`

La línea 29 utiliza una de las diversas funciones **`insert`** sobrecargadas que proporciona cada *contenedor de secuencia* (excepto `array`, que tiene un tamaño fijo, y `forward_list`, que tiene la función `insert_after` en su defecto). En la línea 29 se inserta el valor 22 antes del elemento que se encuentra en la posición especificada por el iterador en el primer argumento. En este ejemplo, el iterador apunta al segundo elemento del `vector`, por lo que se inserta 22 en el segundo elemento y el segundo elemento original se convierte en el tercer elemento. Otras versiones de `insert` permiten insertar varias copias del mismo valor, empezando desde una posición específica en el contenedor, o insertar un rango de valores desde otro contenedor, empezando desde una posición específica. A partir de C++11, esta versión de la función miembro `insert` devuelve un iterador que apunta al elemento que se insertó.

Función miembro `erase` *de* `vector`

Las líneas 45 y 50 utilizan las dos funciones **`erase`** que están disponibles en todos los *contenedores de primera clase* (excepto `array`, que tiene un tamaño fijo, y `forward_list`, que tiene la función `erase_after` en su defecto). La línea 45 borra el elemento en la ubicación especificada por el argumento iterador (en este ejemplo, el primer elemento). La línea 50 especifica que deben eliminarse todos los elementos en el rango especificado por los dos argumentos iteradores. En este ejemplo se eliminan todos los elementos. En la línea 52 se utiliza la función **`empty`** (disponible para todos los contenedores y adaptadores) para confirmar que el `vector` está vacío.

Error común de programación 15.3

Por lo general, `erase` *destruye los objetos que se eliminan de un contenedor. Sin embargo, eliminar un elemento que contiene un apuntador a un objeto asignado en forma dinámica* no *elimina a la memoria asignada en forma dinámica; esto puede provocar una fuga de memoria. Si el elemento es un* `unique_ptr`*, éste se destruiría y se* eliminaría *también la memoria asignada en forma dinámica. Si el elemento es un* `shared_ptr`*, se decrementaría el conteo de referencias al objeto asignado en forma dinámica y la memoria se eliminaría* sólo *si el conteo de referencias llegara a* `0`*.*

Función miembro* `insert` *de* `vector` *con tres argumentos (*`insert` *rango)

En la línea 55 se demuestra la versión de la función `insert` que utiliza al segundo y tercer argumentos para especificar las posiciones inicial y final en una secuencia de valores (en este caso, del arreglo `valores`) que deben insertarse en el `vector`. Recuerde que la posición final especifica la posición en la secuencia que se encuentra *después* del último elemento a insertar; la copia se lleva a cabo hasta (pero *sin* incluir) esta posición. A partir de C++11, esta versión de la función miembro `insert` devuelve un apuntador que apunta al primer elemento que se insertó; si no se insertó nada, la función devuelve su primer argumento.

***Función miembro* `clear` *de* `vector`**

Por último, en la línea 60 se utiliza la función `clear` (disponible en todos los *contenedores de primera clase* excepto `array`) para vaciar el `vector`; esto no necesariamente devuelve la memoria del `vector` al sistema. [*Nota:* en las siguientes secciones veremos muchas funciones miembro de contenedores comunes. También cubriremos muchas funciones específicas para cada contenedor].

15.5.2 Contenedor de secuencia `list`

El *contenedor de secuencia* `list` (del encabezado `<list>`) permite operaciones de inserción y eliminación en *cualquier* posición del contenedor. Si la mayoría de las inserciones y eliminaciones ocurren en los *extremos* del contenedor, la estructura de datos `deque` (sección 15.5.3) ofrece una implementación más eficiente. La plantilla de clase `list` se implementa como una *lista con enlace doble*: cada nodo en la lista contiene un apuntador al nodo anterior y al siguiente nodo de esta lista. Esto permite a la plantilla de clase `list` soportar *iteradores bidireccionales* que permitan que el contenedor se recorra tanto hacia delante como hacia atrás. Cualquier algoritmo que requiera iteradores de *entrada*, *salida*, *de avance* o *bidireccionales* puede operar sobre un contenedor `list`. Muchas de las funciones miembro de este contenedor manipulan a sus elementos como un conjunto ordenado.

***C++11: contenedor* `forward_list`**

Ahora C++11 incluye el nuevo contenedor de secuencia `forward_list` (encabezado `<forward_list>`), que se implementa como una *lista con enlace simple*: cada nodo en la lista contiene un apuntador al siguiente nodo de la misma. Esto permite a la plantilla de clase `list` soportar *iteradores de avance* que permiten recorrer el contenedor en dirección hacia delante. Cualquier algoritmo que requiera *operadores de entrada*, *salida* o *de avance* puede operar en un contenedor `forward_list`.

***Funciones miembro de* `list`**

Además de las funciones miembro de la figura 15.2 y de las funciones miembro comunes de todos los *contenedores de secuencia* descritas en la sección 15.5, la plantilla de clase `list` cuenta con otras nueve funciones miembro: `splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` y `sort`. Varias de estas funciones miembro son implementaciones optimizadas en el contenedor `list` de

los algoritmos de la Biblioteca estándar que se presentan en el capítulo 16. Tanto `push_front` como `pop_front` son soportadas también por `forward_list` y `deque`. La figura 15.13 demuestra varias características de la clase `list`. Recuerde que muchas de las funciones presentadas en las figuras de la 15.10 a la 15.11 pueden utilizarse con la clase `list`, por lo que nos enfocaremos en las nuevas características en la discusión de este ejemplo.

```
1  // Fig. 15.13: fig15_13.cpp
2  // Plantilla de clase list de la Biblioteca estándar.
3  #include <iostream>
4  #include <array>
5  #include <list> // definición de la plantilla de clase list
6  #include <algorithm> // algoritmo de copia
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 // prototipo para la plantilla de función imprimirLista
11 template < typename T > void imprimirLista( const list< T > &refLista );
12
13 int main()
14 {
15    const size_t TAMANIO = 4;
16    array< int, TAMANIO > enteros = { 2, 6, 4, 8 };
17    list< int > valores; // crea una lista de valores int
18    list< int > otrosValores; // crea una lista de valores int
19
20    // inserta elementos en valores
21    valores.push_front( 1 );
22    valores.push_front( 2 );
23    valores.push_back( 4 );
24    valores.push_back( 3 );
25
26    cout << "valores contiene: ";
27    imprimirLista( valores );
28
29    valores.sort(); // ordena los valores
30    cout << "\nvalores despues de ordenar contiene: ";
31    imprimirLista( valores );
32
33    // inserta elementos de enteros en otrosValores
34    otrosValores.insert( otrosValores.cbegin(), enteros.cbegin(), enteros.cend() );
35    cout << "\nDespues de insert, otrosValores contiene: ";
36    imprimirLista( otrosValores );
37
38    // elimina los elementos de otrosValores e inserta al final de valores
39    valores.splice( valores.cend(), otrosValores );
40    cout << "\nDespues de splice, valores contiene: ";
41    imprimirLista( valores );
42
43    valores.sort(); // ordena valores
44    cout << "\nDespues de sort, valores contiene: ";
45    imprimirLista( valores );
46
```

Fig. 15.13 | Plantilla de clase `list` de la Biblioteca estándar (parte 1 de 3).

```
47     // inserta elementos de enteros en otrosValores
48     otrosValores.insert( otrosValores.cbegin(), enteros.cbegin(), enteros.cend() );
49     otrosValores.sort(); // ordena la lista
50     cout << "\nDespues de insert y sort, otrosValores contiene: ";
51     imprimirLista( otrosValores );
52
53     // elimina los elementos de otrosValores y los inserta en valores por orden
54     valores.merge( otrosValores );
55     cout << "\nDespues de merge:\n   valores contiene: ";
56     imprimirLista( valores );
57     cout << "\n   otrosValores contiene: ";
58     imprimirLista( otrosValores );
59
60     valores.pop_front(); // elimina elemento de la parte inicial
61     valores.pop_back(); // elimina elemento de la parte final
62     cout << "\nDespues de pop_front y pop_back:\n   valores contiene: "
63     imprimirLista( valores );
64
65     valores.unique(); // elimina elementos duplicados
66     cout << "\nDespues de unique, valores contiene: ";
67     imprimirLista( valores );
68
69     // intercambia los elementos de valores y otrosValores
70     valores.swap( otrosValores );
71     cout << "\nDespues de swap:\n   valores contiene: ";
72     imprimirLista( valores );
73     cout << "\n   otrosValores contiene: ";
74     imprimirLista( otrosValores );
75
76     // reemplaza el contenido de valores con elementos de otrosValores
77     valores.assign( otrosValores.cbegin(), otrosValores.cend() );
78     cout << "\nDespues de assign, valores contiene: ";
79     imprimirLista( valores );
80
81     // elimina los elementos de otrosValores y los inserta en valores por orden
82     valores.merge( otrosValores );
83     cout << "\nDespues de merge, valores contiene: ";
84     imprimirLista( valores );
85
86     valores.remove( 4 ); // elimina todos los 4s
87     cout << "\nDespues de remove( 4 ), valores contiene: ";
88     imprimirLista( valores );
89     cout << endl;
90   } // fin de main
91
92   // definición de la plantilla de función imprimirLista; usa
93   // ostream_iterator y algoritmo copy para mostrar los elementos de la lista
94   template < typename T > void imprimirLista( const list< T > &refLista )
95   {
96     if ( refLista.empty() ) // lista está vacía
97       cout << "Lista esta vacia";
98     else
99     {
```

Fig. 15.13 | Plantilla de clase `list` de la Biblioteca estándar (parte 2 de 3).

```
100       ostream_iterator< T > salida( cout, " " );
101       copy( refLista.cbegin(), refLista.cend(), salida );
102    } // fin de else
103 } // fin de la función imprimirLista
```

```
valores contiene: 2 1 4 3
valores despues de ordenar contiene: 1 2 3 4
Despues de insert, otrosValores contiene: 2 6 4 8
Despues de splice, valores contiene: 1 2 3 4 2 6 4 8
Despues de sort, valores contiene: 1 2 2 3 4 4 6 8
Despues de insert y sort, otrosValores contiene: 2 4 6 8
Despues de merge:
   valores contiene: 1 2 2 2 3 4 4 4 6 6 8 8
   otrosValores contiene: Lista esta vacia
Despues de pop_front y pop_back:
   valores contiene: 2 2 2 3 4 4 4 6 6 8
Despues de unique, valores contiene: 2 3 4 6 8
Despues de swap:
   valores contiene: Lista esta vacia
   otrosValores contiene: 2 3 4 6 8
Despues de assign, valores contiene: 2 3 4 6 8
Despues de merge, valores contiene: 2 2 3 3 4 4 6 6 8 8
Despues de remove( 4 ), valores contiene: 2 2 3 3 6 6 8 8
```

Fig. 15.13 | Plantilla de clase `list` de la Biblioteca estándar (parte 3 de 3).

Creación de objetos `list`

En las líneas 17 y 18 se crean instancias de dos objetos `list` capaces de almacenar valores `int`. Las líneas 21 y 22 usan a la función **`push_front`** para insertar enteros al inicio de `valores`. La función `push_front` es específica para las clases `forward_list`, `list` y `deque`. Las líneas 23 y 24 utilizan la función `push_back` para insertar enteros al final de `valores`. *La función `push_back` es común para todos los contenedores de secuencia*, excepto `array` y `forward_list`.

Función miembro `sort` *de* `list`

La línea 29 utiliza la función miembro **`sort`** de `list` para ordenar los elementos en *forma ascendente*. [*Nota*: esta función es distinta a la función `sort` en los algoritmos de la Biblioteca estándar]. Hay una segunda versión de `sort` que permite al programador suministrar una *función predicado binaria* que toma dos argumentos (valores en la lista), realiza una comparación y devuelve un valor `bool` indicando si el primer argumento debe ir antes del segundo en el contenido almacenado. Esta función determina el orden en el que se ordenan los elementos del contenedor `list`. Esta versión podría ser especialmente útil para un contenedor `list` que almacene apuntadores en vez de valores. [*Nota*: en la figura 16.3 demostramos el uso de una *función predicado unaria*. Este tipo de función toma un solo argumento, realiza una comparación utilizando ese argumento y devuelve un valor `bool` indicando el resultado].

Función miembro `splice` *de* `list`

La línea 39 utiliza la función **`splice`** de `list` para eliminar los elementos en `otrosValores` e insertarlos en `valores` antes de la posición del iterador especificado como el primer argumento. Hay otras dos versiones de esta función. La función `splice` con tres argumentos permite que se elimine un elemento del contenedor especificado como el segundo argumento y desde la posición especificada por el iterador

en el tercer argumento. La función `splice` con cuatro argumentos utiliza los últimos dos para especificar un rango de posiciones que deben eliminarse del contenedor en el segundo argumento, y colocarse en la ubicación especificada en el primer argumento. La plantilla de clase `forward_list` cuenta con una función miembro similar, llamada `splice_after`.

Función miembro **`merge`** *de* **`list`**
Después de insertar más elementos en la lista `otrosValores` y de *ordenar* tanto a `valores` como a `otrosValores`, en la línea 54 se utiliza la función miembro **`merge`** de `list` para eliminar a todos los elementos de `otrosValores` e insertarlos en forma ordenada en `valores`. Ambas listas deben *ordenarse* en el *mismo* orden antes de realizar esta operación. Una segunda versión de `merge` permite al programador suministrar una *función predicado binaria* que toma dos argumentos (valores en la lista) y devuelve un valor `bool`. La función predicado especifica el orden utilizado por `merge`.

Función miembro **`pop_front`** *de* **`list`**
En la línea 60 se utiliza la función **`pop_front`** de `list` para eliminar el primer elemento en la lista. En la línea 60 se utiliza la función **`pop_back`** (disponible para todos los *contenedores de secuencia* excepto `array` y `forward_list`) para eliminar el último elemento en la lista.

Función miembro **`unique`** *de* **`list`**
En la línea 65 se utiliza la función **`unique`** de `list` para *eliminar los elementos duplicados* en la lista. Ésta debe estar *ordenada* (de manera que todos los duplicados estén uno al lado del otro) antes de realizar esta operación, para garantizar que se eliminen todos los duplicados. Una segunda versión de `unique` permite al programador suministrar una *función predicado* que toma dos argumentos (valores en la lista) y devuelve un valor `bool` que especifica si dos elementos son iguales.

Función miembro **`swap`** *de* **`list`**
En la línea 70 se utiliza la función **`swap`** (disponible para todos los *contenedores de primera clase*) para intercambiar el contenido de `valores` con el contenido de `otrosValores`.

Funciones miembro **`assign`** *y* **`remove`** *de* **`list`**
En la línea 77 se utiliza la función **`assign`** de `list` (disponible para todos los *contenedores de secuencia*) para reemplazar el contenido de `valores` con el contenido de `otrosValores`, en el rango especificado por los dos argumentos iteradores. Una segunda versión de `assign` reemplaza el contenido original con copias del valor especificado en el segundo argumento. El primer argumento de la función especifica el número de copias. En la línea 86 se utiliza la función **`remove`** de `list` para eliminar todas las copias del valor 4 de la lista.

15.5.3 Contenedor de secuencia deque

La clase **`deque`** ofrece muchos de los beneficios de `vector` y `list` en un contenedor. El término en inglés `deque` es la abreviación de "cola con dos partes finales". La clase `deque` está implementada para ofrecer un acceso indizado eficiente (mediante los subíndices) para leer y modificar sus elementos, en forma muy parecida a un `vector`. La clase `deque` también está implementada para producir *operaciones eficientes de inserción y eliminación en su parte inicial y final*, en forma muy parecida a un contenedor `list` (aunque `list` es también capaz de realizar inserciones y eliminaciones eficientemente a la *mitad* de un contenedor `list`). La clase `deque` ofrece soporte para los iteradores de acceso aleatorio, por lo que puede utilizarse con todos los algoritmos de la Biblioteca estándar. Uno de los usos más comunes de un contenedor `deque` es el de mantener una *cola* de elementos *del tipo "primero en entrar, primero en salir"*. De hecho, `deque` es la implementación subyacente predeterminada para el adaptador `queue` (sección 15.7.2).

Puede asignarse más espacio de almacenamiento para un contenedor `deque` en cualquiera de sus extremos, en bloques de memoria que se mantienen generalmente como un arreglo integrado de apuntadores a esos bloques.[2] Debido a la *distribución de memoria no contigua* de un contenedor `deque`, un iterador para este tipo de contenedores debe ser más "inteligente" que los apuntadores utilizados para iterar a través de contenedores `vector`, `array` o arreglos basados en apuntadores.

Tip de rendimiento 15.8

En general, `deque` tiene mayor sobrecarga que `vector`.

Tip de rendimiento 15.9

Las inserciones y eliminaciones a la mitad de un contenedor `deque` se optimizan para minimizar el número de elementos copiados, por lo que es más eficiente que un contenedor `vector` pero menos eficiente que un contenedor `list` para este tipo de modificación.

La clase `deque` cuenta con las mismas operaciones básicas que la clase `vector` pero, al igual que `list`, agrega las funciones miembro **`push_front`** y **`pop_front`** para permitir la inserción y eliminación al principio del contenedor `deque`, respectivamente.

La figura 15.14 demuestra las características de la clase `deque`. Recuerde que muchas de las funciones presentadas en las figuras 15.10, 15.11 y 15.13 también pueden utilizarse con la clase `deque`. Debe incluirse el encabezado **`<deque>`** para poder utilizar esta clase.

```
1  // Fig. 15.14: fig15_14.cpp
2  // Plantilla de clase deque de la Biblioteca estándar.
3  #include <iostream>
4  #include <deque> // definición de la plantilla de clase deque
5  #include <algorithm> // algoritmo de copia
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  int main()
10 {
11    deque< double > valores; // crea deque de valores double
12    ostream_iterator< double > salida( cout, " " );
13
14    // inserta elementos en valores
15    valores.push_front( 2.2 );
16    valores.push_front( 3.5 );
17    valores.push_back( 1.1 );
18
19    cout << "valores contiene: ";
20
21    // usa el operador subíndice para obtener elementos de valores
22    for ( size_t i = 0; i < valores.size() ; ++i )
23       cout << valores[ i ] << ' ';
24
25    valores.pop_front(); // elimina el primer elemento
26    cout << "\nDespues de pop_front, valores contiene: ";
27    copy( valores.cbegin(), valores.cend(), salida );
28
```

Fig. 15.14 | Plantilla de clase `deque` de la Biblioteca estándar (parte 1 de 2).

2 Este detalle es específico de cada implementación, no un requerimiento del estándar de C++.

```
29     // usa el operador subíndice para modificar el elemento en la ubicación 1
30     valores[ 1 ] = 5.4;
31     cout << "\nDespues de valores[ 1 ] = 5.4, valores contiene: ";
32     copy( valores.cbegin(), valores.cend(), salida );
33     cout << endl;
34  } // fin de main
```

```
valores contiene: 3.5 2.2 1.1
Despues de pop_front, valores contiene: 2.2 1.1
Despues de valores[ 1 ] = 5.4, valores contiene: 2.2 5.4
```

Fig. 15.14 | Plantilla de clase `deque` de la Biblioteca estándar (parte 2 de 2).

En la línea 11 se crea una instancia de un contenedor `deque` que puede almacenar valores `double`. En las líneas 15 a 17 se utilizan las funciones `push_front` y `push_back` para insertar elementos en la parte inicial y final del contenedor `deque`.

La instrucción `for` en las líneas 22 y 23 utiliza el operador subíndice para recuperar el valor en cada elemento del contenedor `deque` y así poder mostrarlo. La condición utiliza a la función `size` para asegurar que no tratemos de utilizar un elemento *fuera de* los límites del contenedor `deque`.

En la línea 25 se utiliza la función `pop_front` para demostrar cómo eliminar el primer elemento del contenedor `deque`. En la línea 30 se utiliza el operador de subíndice para obtener un *lvalue*. Esto permite que los valores se asignen directamente a cualquier elemento del contenedor `deque`.

15.6 Contenedores asociativos

Los *contenedores asociativos* de la Biblioteca estándar ofrecen un *acceso directo* para almacenar y recuperar elementos mediante **claves** (comúnmente conocidas como **claves de búsqueda**). Los cuatro *contenedores asociativos ordenados* son `multiset`, `set`, `multimap` y `map`. Cada uno de éstos mantiene sus claves *en orden*. También hay cuatro *contenedores asociativos desordenados* correspondientes: `unordered_multiset`, `unordered_set`, `unordered_multimap` y `unordered_map`, los cuales ofrecen la mayoría de las mismas capacidades que sus contrapartes ordenados. La principal diferencia entre los contenedores asociativos ordenados y desordenados es que los desordenados *no* mantienen sus claves *en orden*. En esta sección nos concentraremos en los *contenedores asociativos ordenados*.

Tip de rendimiento 15.10

Los contenedores asociativos desordenados podrían ofrecer un mejor rendimiento en casos en los que no es necesario mantener las claves en orden.

Al iterar a través de un *contenedor asociativo ordenado*, éste se recorre en el orden asignado a ese contenedor. Las clases **`multiset`** y **`set`** cuentan con operaciones para manipular conjuntos de valores en donde éstos son las claves; *no* hay un valor separado asociado con cada clave. La principal diferencia entre `multiset` y `set` es que `multiset` *permite claves duplicadas* y `set` no. Las clases **`multimap`** y **`map`** ofrecen operaciones para manipular valores asociados con claves (estos valores algunas veces se conocen como **valores asignados**). La principal diferencia entre `multimap` y `map` es que `multimap` permite *claves duplicadas* con los valores asociados que van a almacenarse y `map` solamente permite *claves únicas* con los valores asociados. Además de las funciones miembro comunes de todos los contenedores, los *contenedores asociativos ordenados* también soportan otras funciones miembro que son específicas para los conte-

nedores asociativos. En las siguientes subsecciones se presentan ejemplos de cada uno de los *contenedores asociativos ordenados* y las funciones miembro comunes para ellos.

15.6.1 Contenedor asociativo `multiset`

El *contenedor asociativo ordenado* `multiset` (del encabezado **<set>**) ofrece rapidez en el almacenamiento y la recuperación de las claves, además de que permite el uso de claves duplicadas. El ordenamiento de los elementos se determina mediante un **objeto función de comparación**. Por ejemplo, en un `multiset` entero, los elementos pueden ordenarse en *forma ascendente* si se ordenan las claves con el **objeto función de comparación `less< int >`**. En la sección 16.4 hablaremos con detalle sobre los objetos función. En este capítulo sólo mostraremos cómo usar `less<int>` cuando se declaran contenedores asociativos ordenados. El tipo de datos de las claves en todos los *contenedores asociativos ordenados* debe soportar la comparación, basándose en el objeto función de comparación; las claves ordenadas con `less< T >` deben soportar la comparación con `operator<`. Si las claves utilizadas en los *contenedores asociativos ordenados* son de tipos de datos definidos por el usuario, esos tipos deben proporcionar los operadores de comparación apropiados. Un contenedor `multiset` soporta los *iteradores bidireccionales* (pero no los *iteradores de acceso aleatorio*). Y si el orden de las claves no es importante, puede usar `unordered_multiset` (encabezado <unordered_set>) como alternativa.

La figura 15.15 demuestra el uso del *contenedor asociativo ordenado* `multiset` para un `multiset` de enteros con claves ordenadas en *forma ascendente*. Los contenedores `multiset` y `set` (sección 15.6.2) cuentan con la misma funcionalidad básica.

```
1  // Fig. 15.15: fig15_15.cpp
2  // Plantilla de clase multiset de la Biblitoeca estándar
3  #include <array>
4  #include <iostream>
5  #include <set> // definición de la plantilla de clase multiset
6  #include <algorithm> // algoritmo de copia
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 int main()
11 {
12    const size_t TAMANIO = 10;
13    array< int, TAMANIO > a = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
14    multiset< int, less< int > > multisetInt; // multiset de enteros
15    ostream_iterator< int > salida( cout, " " );
16
17    cout << "Hay actualmente " << multisetInt.count( 15 )
18       << " valores de 15 en el multiset\n";
19
20    multisetInt.insert( 15 ); // inserta 15 en multisetInt
21    multisetInt.insert( 15 ); // inserta 15 en multisetInt
22    cout << "Despues de insert, hay " << multisetInt.count( 15 )
23       << " valores de 15 en el multiset\n\n";
24
25    // busca 15 en multisetInt; find devuelve el iterador
26    auto resultado = multisetInt.find( 15 );
27
28    if ( resultado != multisetInt.end() ) // si el iterador no está al final
29       cout << "Se encontro el valor 15\n"; // encontró el valor de búsqueda 15
```

Fig. 15.15 | Plantilla de clase `multiset` de la Biblioteca estándar (parte 1 de 2).

```
30
31     // busca 20 en multisetInt; find devuelve el iterador
32     resultado = multisetInt.find( 20 );
33
34     if ( resultado == multisetInt.end() ) // será verdadero, ya que
35        cout << "No se encontro el valor 20\n"; // no encontró el 20
36
37     // inserta los elementos del arreglo a en multisetInt
38     multisetInt.insert( a.cbegin(), a.cend() );
39     cout << "\nDespues de insert, multisetInt contiene:\n";
40     multisetInt.begin(), multisetInt.end(), salida );
41
42     // determina los límites inferior y superior de 22 en multisetInt
43     cout << "\n\nLimite inferior de 22: "
44        << *( multisetInt.lower_bound( 22 ) ) ;
45     cout << "\nLimite superior de 22: " << *( multisetInt.upper_bound( 22 ) ) ;
46
47     // usa equal_range para determinar los límites inferior y superior
48     // de 22 en multisetInt
49     auto p = multisetInt.equal_range( 22 );
50
51     cout << "\n\nequal_range de 22:" << "\n  Limite inferior: "
52        << *( p.first ) << "\n  Limite superior: " << *( p.second ) ;
53     cout << endl;
54  } // fin de main
```

```
Hay actualmente 0 valores de 15 en el multiset
Despues de insert, hay 2 valores de 15 en el multiset

Se encontro el valor 15
No se encontro el valor 20

Despues de insert, multisetInt contiene:
1 7 9 13 15 15 18 22 22 30 85 100

Limite inferior de 22: 22
Limite superior de 22: 30

equal_range de 22:
   Limite inferior: 22
   Limite superior: 30
```

Fig. 15.15 | Plantilla de clase `multiset` de la Biblioteca estándar (parte 2 de 2).

Creación de un `multiset`

La línea 14 crea un `multiset` de enteros ordenados en *forma ascendente*, utilizando el objeto función `less<int>`. El *orden ascendente* es el valor predeterminado para un conjunto `multiset`, por lo que se puede omitir `less<int>`. C++11 corrige un problema del compilador con el espacio entre el > que cierra `less<int>` y el > que cierra el tipo `multiset`. Antes de C++11, si se especificaba este tipo de como

```
multiset<int, less<int>> multisetInt;
```

el compilador trataría los caracteres >> al final del tipo como el operador >> y generaría un error de compilación. Por esta razón era necesario colocar un espacio entre el > que cierra `less<int>` y el > que

cierra el tipo `multiset` (o cualquier otro tipo de plantilla similar, como `vector<vector<int>>`). A partir de C++11, la declaración anterior se compila correctamente.

Función miembro `const` *de* `multiset`

En la línea 17 se utiliza la función **`count`** (disponible para todos los *contenedores asociativos*) para contar el número de ocurrencias del valor `15` que haya actualmente en el `multiset`.

Función miembro `insert` *de* `multiset`

En las líneas 20 y 21 se utiliza una de las diversas versiones sobrecargadas de la función `insert` para agregar el valor `15` al `multiset` dos veces. Una segunda versión de `insert` toma un iterador y un valor como argumentos, y empieza la búsqueda del punto de inserción para la posición especificada del iterador. Una tercera versión de `insert` toma dos iteradores como argumentos, que especifican un rango de valores a agregar al `multiset` provenientes de otro contenedor.

Función miembro `find` *de* `multiset`

En la línea 26 se utiliza a la función **`find`** (disponible para todos los *contenedores asociativos*) para localizar el valor `15` en el `multiset`. La función `find` devuelve un `iterator` o un `const_iterator` que apunta a la ubicación en la que se encontró el valor. Si éste *no* se encuentra, `find` devuelve un `iterator` o un `const_iterator` igual al valor devuelto por una llamada a `end` en el contenedor. La línea 32 demuestra este caso.

Insertar elementos de otro contenedor en un `multiset`

En la línea 38 se utiliza la función **`insert`** para insertar los elementos del arreglo `a` en el `multiset`. En la línea 40, el algoritmo `copy` copia los elementos del `multiset` a la salida estándar en *orden ascendente*.

Funciones miembro `lower_bound` *y* `upper_bound` *de* `multiset`

En las líneas 44 y 45 se utilizan las funciones **`lower_bound`** y **`upper_bound`** (disponibles en todos los *contenedores asociativos*) para localizar la primera ocurrencia del valor `22` en el `multiset` y el elemento que está *después* de la primera ocurrencia del valor `22` en el `multiset`. Ambas funciones devuelven iteradores tipo `iterator` o `const_iterator` que apuntan a la posición apropiada del iterador devuelto por `end`, en caso de que el valor no se encuentre en el contenedor `multiset`.

Los objetos `pair` *y la función miembro* `equal_range` *de* `multiset`

En la línea 49 se crea e inicializa un objeto `pair` llamado `p`. Una vez más, usamos la palabra clave `auto` de C++11 para inferir el tipo de la variable de su inicializador; en este caso, el valor de retorno de la función miembro `equal_range` de `multiset`, que es un objeto `pair`. Dichos objetos asocian pares de valores. El contenido de un objeto `p` está compuesto de dos iteradores del tipo `const_iterator` para nuestro `multiset` de enteros. La función **`equal_range`** de `multiset` devuelve un objeto `pair` que contiene los resultados de llamar a `lower_bound` y `upper_bound`. El tipo `pair` contiene dos miembros de datos `public` llamados **`first`** y **`second`**. En la línea 49 se utiliza la función `equal_range` para determinar el límite inferior (`lower_bound`) y el límite superior (`upper_bound`) de `22` en el contenedor `multiset`. En la línea 52 se utiliza `p.first` y `p.second` para tener acceso a los límites menor (`lower_bound`) e inferior (`upper_bound`). *Desreferenciamos* a los iteradores para mostrar los valores en las posiciones devueltas por `equal_range`. Aunque no lo hicimos aquí, debemos siempre asegurarnos de que los iteradores devueltos por `lower_bound`, `upper_bound` y `equal_range` no sean iguales al iterador *final* del contenedor antes de desreferenciar a los iteradores.

C++11: Plantilla de clase variádica **`tuple`**
C++ también incluye la plantilla de clase `tuple`, que es similar a `pair`, sólo que puede contener cualquier número de elementos de diversos tipos. A partir de C++11, la plantilla de clase `tuple` se reimplementó mediante el uso de *plantillas variádicas*: plantillas que pueden recibir un número *variable* de argumentos. En el capítulo 24, C++11: Additional Features, hablaremos sobre `tuple` y las plantillas variádicas.

15.6.2 Contenedor asociativo `set`

El *contenedor asociativo* `set` (del encabezado `<set>`) se utiliza para almacenar y recuperar rápidamente claves únicas. La implementación de `set` es idéntica a la de `multiset`, sólo que un contenedor `set` debe tener claves únicas. Por lo tanto, si se trata de insertar una clave *duplicada* en un contenedor `set`, se ignora ese duplicado; como éste es el comportamiento matemático deseado para un conjunto, no lo identificamos como un error de programación común. Un contenedor `set` soporta *iteradores bidireccionales* (pero no los *iteradores de acceso aleatorio*). Si el orden de las claves no es importante, podemos usar `unordered_set` (encabezado `<unordered_set>`) como alternativa. La figura 15.16 demuestra el uso de un contenedor `set` con valores `double`.

```
1  // Fig. 15.16: fig15_16.cpp
2  // Plantilla de clase set de la Biblioteca estándar.
3  #include <iostream>
4  #include <array>
5  #include <set>
6  #include <algorithm>
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 int main()
11 {
12    const size_t TAMANIO = 5;
13    array< double, TAMANIO > a = { 2.1, 4.2, 9.5, 2.1, 3.7 };
14    set< double, less< double > > setDouble( a.begin(), a.end() );
15    ostream_iterator< double > salida( cout, " " );
16
17    cout << "setDouble contiene: ";
18    copy( setDouble.begin(), setDouble.end(), salida );
19
20    // inserta 13.8 en setDouble; insert devuelve un par en el cual
21    // p.first representa la ubicación de 13.8 en setDouble y
22    // p.second representa si se insertó o no el valor 13.8
23    auto p = setDouble.insert( 13.8 ); // el valor no está en el conjunto
24    cout << "\n\n" << *( p.first )
25       << ( p.second ? " se" : " no se" ) << " inserto";
26    cout << "\ndoubleSet contiene: ";
27    copy( setDouble.begin(), setDouble.end(), salida );
28
29    // inserta 9.5 en setDouble
30    p = setDouble.insert( 9.5 ); // el valor ya está en el conjunto
31    cout << "\n\n" << *( p.first )
32       << ( p.second ? " se" : " no se" ) << " inserto";
33    cout << "\ndoubleSet contiene: ";
```

Fig. 15.16 | Plantilla de clase `set` de la Biblioteca estándar (parte 1 de 2).

```
34     copy( setDouble.begin(), setDouble.end(), salida );
35     cout << endl;
36  } // fin de main
```

```
setDouble contiene: 2.1 3.7 4.2 9.5

13.8 se inserto
doubleSet contiene: 2.1 3.7 4.2 9.5 13.8

9.5 no se inserto
doubleSet contiene: 2.1 3.7 4.2 9.5 13.8
```

Fig. 15.16 | Plantilla de clase `set` de la Biblioteca estándar (parte 2 de 2).

En la línea 14 se crea un conjunto `set` de valores `double` ordenados en *forma ascendente*, utilizando el objeto función `less<double>`. La llamada al constructor toma los elementos en el arreglo `a` y los inserta en el contenedor `set`. En la línea 18 se utiliza el algoritmo `copy` para mostrar el contenido del conjunto. Observe que el valor `2.1`, que aparece dos veces en el arreglo `a`, aparece sólo *una vez* en `setDouble`. Esto se debe a que el contenedor `set` *no* permite duplicados.

En la línea 23 se define e inicializa un objeto `pair` para almacenar el resultado de una llamada a la función `insert` de `set`. El objeto `pair` devuelto, consiste en un iterador `const_iterator` que apunta al elemento en el contenedor `set` que se insertó y un valor `bool` que indica si se insertó o no el elemento; `true` si el elemento no estaba en el set; `false` si estaba.

En la línea 23 se utiliza la función `insert` para colocar el valor `13.8` en el contenedor `set`. El objeto `pair` devuelto, llamado `p`, contiene un iterador `p.first` que apunta al valor `13.8` en el contenedor `set` y un valor `bool` que es `true` debido a que el valor se insertó. En la línea 30 se hace un intento por insertar `9.5`, que ya se encuentra en el conjunto. La salida muestra que `9.5` no se insertó de nuevo, debido a que los conjuntos no permiten claves duplicadas. En este caso, `p.first` en el objeto `pair` devuelto apunta al `9.5` existente en el conjunto `set`.

15.6.3 Contenedor asociativo `multimap`

El *contenedor asociativo* **`multimap`** se utiliza para almacenar y recuperar rápidamente las claves y los valores asociados (a menudo conocidos como pares clave/valor). Muchas de las funciones utilizadas con los contenedores `multiset` y `set` también se utilizan con `multimap` y `map`. Los elementos de los contenedores `multimap` y `map` son pares (objetos de la clase `pair`) de claves y valores, en vez de valores individuales. Al insertar datos en un `multimap` o `map` se utiliza un objeto `pair` que contiene la clave y el valor. El orden de las claves se determina mediante un *objeto función de comparación*. Por ejemplo, en un contenedor `multimap` que utilice enteros como el tipo para las claves, éstas pueden ordenarse en *forma ascendente* mediante el *objeto función de comparación* `less<int>`. En un contenedor `multimap` se permiten claves duplicadas, por lo que pueden asociarse varios valores con una sola clave. Esto a menudo se conoce como una **relación de uno a varios**. Por ejemplo, en un sistema para procesar transacciones de tarjetas de crédito, la cuenta de una tarjeta de crédito puede tener muchas transacciones asociadas; en una universidad, un estudiante puede tomar muchas clases y un profesor puede enseñar a muchos estudiantes; en la milicia, un rango (como "cabo") tiene muchas personas. Un contenedor `multimap` soporta el uso de *iteradores bidireccionales*, pero no los *iteradores de acceso aleatorio*. En la figura 15.17 se demuestra el uso del *contenedor asociativo* `multimap`. Debe incluirse el encabezado **`<map>`** para poder utilizar la clase `multimap`. Si el orden de las claves no es importante, podemos usar `unordered_multimap` (encabezado `<unordered_multimap>`) como alternativa.

Tip de rendimiento 15.11

Un contenedor `multimap` se implementa para localizar eficientemente todos los valores que forman pares con una clave dada.

```
1  // Fig. 15.17: fig15_17.cpp
2  // Plantilla de clase multimap de la Biblioteca estándar.
3  #include <iostream>
4  #include <map> // definición de la plantilla de clase multimap
5  using namespace std;
6
7  int main()
8  {
9     multimap< int, double, less< int > > pares; // crea multimap
10
11    cout << "Hay actualmente " << pares.count( 15 )
12       << " pares con la clave 15 en el multimap\n";
13
14    // inserta dos objetos value_type en pares
15    pares.insert( make_pair( 15, 2.7 ) );
16    pares.insert( make_pair( 15, 99.3 ) );
17
18    cout << "Despues de las inserciones, hay " << pares.count( 15 )
19       << " pares con la clave 15\n\n";
20
21    // inserta cinco objetos value_type en pares
22    pares.insert( make_pair( 30, 111.11 ) );
23    pares.insert( make_pair( 10, 22.22 ) );
24    pares.insert( make_pair( 25, 33.333 ) );
25    pares.insert( make_pair( 20, 9.345 ) );
26    pares.insert( make_pair( 5, 77.54 ) );
27
28    cout << "Los pares de multimap contienen:\nClave\tValor\n";
29
30    // recorre los elementos de pares
31    for ( auto mapItem : pares )
32       cout << mapItem.first << '\t'  << mapItem.second << '\n';
33
34    cout << endl;
35 } // fin de main
```

```
Hay actualmente 0 pares con la clave 15 en el multimap
Despues de las inserciones, hay 2 pares con la clave 15

Los pares de multimap contienen:
Clave   Valor
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

Fig. 15.17 | Plantilla de clase `multimap` de la Biblioteca estándar.

En la línea 9 se crea un `multimap` en el que el tipo de la clave es `int`, el tipo del valor asociado con una clave es `double` y los elementos se ordenan en *forma ascendente*. En la línea 11 se utiliza la función `count` para determinar el número de pares clave/valor con una clave de `15` (ninguno aún, ya que el contenedor está vacío actualmente).

En la línea 15 se utiliza la función `insert` para agregar un nuevo par clave/valor al contenedor `multimap`. La expresión `make_pair( 15, 2.7 )` crea un objeto `pair` en el que `first` es la clave (`15`) de tipo `int` y `second` es el valor (`2.7`) de tipo `double`. La función `make_pair` utiliza de manera automática los tipos especificados por el programador para las claves y valores en la declaración del contenedor `multimap` (línea 9). En la línea 16 se inserta otro objeto `pair` con la clave `15` y el valor `99.3`. Después, en las líneas 18 y 19 se muestra el número de pares con la clave `15`. A partir de C++11, es posible usar inicialización con listas para los objetos `pair`, por lo que la línea 15 puede simplificarse como

```
pares.insert( { 15, 2.7 } );
```

De manera similar, C++11 nos permite usar inicialización con listas para inicializar un objeto que se va a devolver de una función. Por ejemplo, si una función devuelve un `pair` que contenga un `int` y un `double`, podría escribir:

```
return { 15, 2.7 };
```

En las líneas 22 a 26 se insertan cinco pares adicionales en el contenedor `multimap`. La instrucción `for` basada en rango de las líneas 31 y 32 muestra el contenido del `multimap`, incluyendo claves y valores. Inferimos el tipo de la variable de control de ciclo (un `pair` que contiene una clave `int` y un valor `double`) con la palabra clave `auto`. En la línea 32 se accede a los miembros del `pair` actual en cada elemento del contenedor `multimap`. Observe en la salida que las claves aparecen en *orden ascendente*.

C++11: Inicialización con listas de un contenedor de pares clave-valor

En este ejemplo utilizamos llamadas separadas a la función miembro `insert` para colocar pares clave-valor en un contenedor `multimap`. Si conocemos los pares clave-valor de antemano, es posible usar la inicialización con listas al momento de crear el contenedor `multimap`. Por ejemplo, la siguiente instrucción inicializa un contenedor `multimap` con tres pares clave-valor que se representan mediante las sublistas en la lista inicializadora principal:

```
multimap< int, double, less< int > > pares =
   { { 10, 22.22 }, { 20, 9.345 }, { 5, 77.54 } };
```

15.6.4 Contenedor asociativo `map`

El *contenedor asociativo `map`* (del encabezado `<map>`) se utiliza para almacenar y recuperar rápidamente *claves únicas* y los *valores asociados*. *No* se permiten claves duplicadas; sólo puede asociarse un valor con cada clave. Esto se conoce como **asignación de uno a uno**. Por ejemplo, una compañía que utiliza números únicos para los empleados tales como 100, 200 y 300 podría tener un contenedor `map` que asocie los números de los empleados con sus extensiones telefónicas: 4321, 4115 y 5217, respectivamente. Con un contenedor `map` el programador especifica la clave y recibe rápidamente de vuelta los datos asociados. Al proporcionar la clave en el operador subíndice `[]` de un contenedor `map` se localiza el valor asociado con esa clave en el contenedor `map`. Pueden realizarse inserciones y eliminaciones en *cualquier parte* de un `map`. Si el orden de las claves no es importante, podemos usar `unordered_map` (encabezado `<unordered_map>`) como alternativa.

En la figura 15.18 se demuestra el uso del contenedor `map` y se utilizan las mismas características que en la figura 15.17 para demostrar el uso del operador subíndice. En las líneas 27 y 28 se utiliza el operador subíndice de la clase `map`. Cuando el subíndice es una clave que ya se encuentra en el contenedor `map` (línea 27), el operador devuelve una referencia a su valor asociado. Cuando el subíndice es una clave que *no* se encuentra en el contenedor `map` (línea 18), el operador inserta la clave en el contenedor

y devuelve una referencia que puede utilizarse para asociar un valor con esa clave. En la línea 27 se sustituye el valor de la clave `25` (anteriormente `33.333`, según lo especificado en la línea 16) con un nuevo valor: `9999.99`. En la línea 28 se inserta un nuevo par clave/valor (lo que se conoce como **crear una asociación**) en el contenedor `map`.

```
1  // Fig. 15.18: fig15_18.cpp
2  // Plantilla de clase map de la Biblioteca estándar.
3  #include <iostream>
4  #include <map> // definición de la plantilla de clase map
5  using namespace std;
6
7  int main()
8  {
9     map< int, double, less< int > > pares;
10
11    // inserta ocho objetos value_type en pares
12    pares.insert( make_pair( 15, 2.7 ) );
13    pares.insert( make_pair( 30, 111.11 ) );
14    pares.insert( make_pair( 5, 1010.1 ) );
15    pares.insert( make_pair( 10, 22.22 ) );
16    pares.insert( make_pair( 25, 33.333 ) );
17    pares.insert( make_pair( 5, 77.54 ) ); // se ignora el valor duplicado
18    pares.insert( make_pair( 20, 9.345 ) );
19    pares.insert( make_pair( 15, 99.3 ) ); // se ignora el valor duplicado
20
21    cout << "pares contiene:\nClave\tValor\n";
22
23    // recorre los elementos de pares
24    for ( auto mapItem : pares )
25       cout << mapItem.first << '\t'  << mapItem.second << '\n';
26
27    pares[ 25 ] = 9999.99; // usa subíndices para modificar el valor de la clave 25
28    pares[ 40 ] = 8765.43; // usa subíndices para insertar el valor de la clave 40
29
30    cout << "\nDespues de las operaciones de subindices, pares contiene:\nClave\
            tValor\n";
31
32    // usa const_iterator para recorrer los elementos de pares
33    for ( auto mapItem : pares )
34       cout << mapItem.first << '\t' << mapItem.second << '\n';
35
36    cout << endl;
37 } // fin de main
```

```
pares contiene:
Clave   Valor
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11
```

Fig. 15.18 | Plantilla de clase `map` de la Biblioteca estándar (parte 1 de 2).

```
Despues de las operaciones de subindices, pares contiene:
Clave   Valor
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

Fig. 15.18 | Plantilla de clase `map` de la Biblioteca estándar (parte 2 de 2).

15.7 Adaptadores de contenedores

Los tres **adaptadores de contenedores** son: `stack`, `queue` y `priority_queue`. Los adaptadores de contenedores *no* son *contenedores de primera clase*, ya que *no* cuentan con la implementación de la estructura de datos actual en la que los elementos pueden almacenarse, y debido a que los adaptadores *no* soportan el uso de iteradores. El beneficio de una *clase de adaptador* es que el programador puede elegir una estructura de datos subyacente apropiada. Las tres *clases de adaptadores* cuentan con las funciones miembro **`push`** y **`pop`** que insertan apropiadamente un elemento en cada estructura de datos de adaptador y eliminan apropiadamente un elemento de cada estructura de datos de adaptador. Las siguientes subsecciones ofrecen ejemplos de las clases de adaptadores.

15.7.1 Adaptador `stack`

La clase **`stack`** (del encabezado **`<stack>`**) permite realizar inserciones y eliminaciones en un extremo de la estructura de datos subyacente que se conoce como su *parte superior*, por lo que una pila comúnmente se conoce como una estructura de datos del tipo *último en entrar, primero en salir*. Introdujimos las pilas en nuestra discusión de la pila de llamadas a funciones en la sección 6.12. Una pila puede implementarse con cualquiera de los contenedores `vector`, `list` o `deque`. Este ejemplo crea tres pilas de enteros utilizando `vector`, `list` y `deque` como estructura de datos subyacente para representar al adaptador `stack`. De manera predeterminada, una pila se implementa con un `deque`. Las operaciones de la pila son `push` para insertar un elemento en su *parte superior* (lo que se implementa mediante una llamada a la función `push_back` del contenedor subyacente), `pop` para eliminar el elemento *superior* de la pila (lo que se implementa mediante una llamada a la función `pop_back` del contenedor subyacente), **`top`** para obtener una referencia al elemento superior de la pila (lo que se implementa mediante una llamada a la función `back` del contenedor subyacente), `empty` para determinar si la pila está vacía o no (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente) y `size` para obtener el número de elementos en la pila (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente). En el capítulo 19 veremos cómo puede desarrollar su propia plantilla de clase de pila personalizada.

En la figura 15.19 se demuestra el uso de la clase de adaptador `stack`. En las líneas 18, 21 y 24 se crean instancias de tres pilas de enteros. En la línea 18 se especifica una pila de enteros que utiliza el contenedor `deque` predeterminado como su estructura de datos subyacente. En la línea 21 se especifica una pila de enteros que utiliza a un `vector` de enteros como su estructura de datos subyacente. En la línea 24 se especifica a una pila de enteros que utiliza un contendor `list` de enteros como su estructura de datos subyacente.

```
1  // Fig. 15.19: fig15_19.cpp
2  // Clase del adaptador stack de la Biblioteca estándar.
3  #include <iostream>
```

Fig. 15.19 | Clase de adaptador `stack` de la Biblioteca estándar (parte 1 de 3).

```
4  #include <stack> // definición del adaptador stack
5  #include <vector> // definición de la plantilla de clase vector
6  #include <list> // definición de la plantilla de clase list
7  using namespace std;
8
9  // prototipo de la plantilla de función meterElementos
10 template< typename T > void meterElementos( T &refStack );
11
12 // prototipo de la plantilla de función sacarElementos
13 template< typename T > void sacarElementos( T &refStack );
14
15 int main()
16 {
17    // pila con el contenedor deque subyacente predeterminado
18    stack< int > pilaDequeInt;
19
20    // pila con el contenedor vector subyacente
21    stack< int, vector< int > > pilaVectorInt;
22
23    // pila con el contenedor list subyacente
24    stack< int, list< int > > pilaListInt;
25
26    // mete los valores 0 a 9 en cada pila
27    cout << "Metiendo datos en pilaDequeInt: ";
28    meterElementos( pilaDequeInt );
29    cout << "\nMetiendo datos en pilaVectorInt: ";
30    meterElementos( pilaVectorInt );
31    cout << "\nMetiendo datos en pilaListInt: ";
32    meterElementos( pilaListInt );
33    cout << endl << endl;
34
35    // muestra y elimina los elementos de cada pila
36    cout << "Sacando datos de pilaDequeInt: ";
37    sacarElementos( pilaDequeInt );
38    cout << "\nSacando datos de pilaVectorInt: ";
39    sacarElementos( pilaVectorInt );
40    cout << "\nSacando datos de pilaListInt: ";
41    sacarElementos( pilaListInt );
42    cout << endl;
43 } // fin de main
44
45 // mete elementos al objeto pila al que refStack hace referencia
46 template< typename T > void meterElementos( T &refStack )
47 {
48    for ( int i = 0; i < 10; ++i )
49    {
50       refStack.push( i ); // mete elemento en la pila
51       cout << refStack.top() << ' '; // ve (y muestra) el elemento superior
52    } // fin de for
53 } // fin de la función meterElementos
54
```

Fig. 15.19 | Clase de adaptador `stack` de la Biblioteca estándar (parte 2 de 3).

```
55  // saca elementos del objeto pila al que refStack hace referencia
56  template< typename T > void sacarElementos( T &refStack )
57  {
58     while ( !refStack.empty() )
59     {
60        cout << refStack.top() << ' '; // ve (y muestra) el elemento superior
61        refStack.pop(); // elimina el elemento superior
62     } // fin de while
63  } // fin de la función sacarElementos
```

```
Metiendo datos en pilaDequeInt: 0 1 2 3 4 5 6 7 8 9
Metiendo datos en pilaVectorInt: 0 1 2 3 4 5 6 7 8 9
Metiendo datos en pilaListInt: 0 1 2 3 4 5 6 7 8 9
Sacando datos de pilaDequeInt: 9 8 7 6 5 4 3 2 1 0
Sacando datos de pilaVectorInt: 9 8 7 6 5 4 3 2 1 0
Sacando datos de pilaListInt: 9 8 7 6 5 4 3 2 1 0
```

Fig. 15.19 | Clase de adaptador `stack` de la Biblioteca estándar (parte 3 de 3).

La función `meterElementos` (líneas 46 a 53) mete los elementos en cada pila. En la línea 50 se utiliza la función `push` (disponible en cada *clase de adaptador*) para colocar un entero en la parte superior de la pila. En la línea 51 se utiliza la función `top` de `stack` para obtener el elemento *superior* de la pila para mostrarlo en pantalla. La *función `top` no elimina el elemento superior*.

La función `sacarElementos` (líneas 56 a 63) saca los elementos de cada pila. En la línea 60 se utiliza la función `top` de `stack` para obtener el elemento superior de la pila y mostrarlo en pantalla. En la línea 61 se utiliza la función `pop` (disponible en cada *clase de adaptador*) para eliminar el elemento *superior* de la pila. La función `pop` *no* devuelve un valor.

15.7.2 Adaptador queue

Una cola es similar a una *línea de espera*. El elemento que ha estado *más tiempo* en la cola es el que se saca *a continuación*; por ello, a una cola se le conoce como estructura de datos del tipo **primero en entrar, primero en salir** (**PEPS**). La clase **`queue`** (del encabezado **`<queue>`**) permite realizar inserciones en la *parte final* de la estructura de datos subyacente y eliminaciones en la *parte inicial* de la misma. Una cola puede almacenar sus elementos en objetos de los contenedores `list` o `deque` de la Biblioteca estándar. De manera predeterminada, una cola se implementa con `deque`. Las operaciones comunes de un adaptador `queue` son `push` para insertar un elemento en la parte final (lo que se implementa mediante una llamada a la función `push_back` del contenedor subyacente), `pop` para eliminar el elemento en la parte inicial de la cola (lo que se implementa mediante una llamada a la función `pop_front` del contenedor subyacente), **`front`** para obtener una referencia al *primer* elemento en la cola (lo que se implementa mediante una llamada a la función `front` del contenedor subyacente), **`back`** para obtener una referencia al último elemento en la cola (lo que se implementa mediante una llamada a la función `back` del contenedor subyacente), `empty` para determinar si la cola está *vacía* o no (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente) y `size` para obtener el número de elementos en la cola (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente). En el capítulo 19 le mostraremos cómo desarrollar su propia plantilla de clase de cola personalizada.

La figura 15.20 demuestra el uso de la clase de adaptador `queue`. En la línea 9 se crea una instancia de `queue` para almacenar valores `double`. En las líneas 12 a 14 se utiliza la función `push` para agregar elementos a la cola. La instrucción `while` en las líneas 19 a 23 usa la función `empty` (disponible en *todos* los contenedores) para determinar si la cola está vacía o no (línea 19). Mientras haya más elementos en `queue`, la línea 21 utiliza la función `front` de `queue` para leer (pero no eliminar) el primer elemento

en queue para salir. La línea 22 elimina el primer elemento en queue mediante la función pop (disponible en todas las *clases de adaptadores*).

```
1  // Fig. 15.20: fig15_20.cpp
2  // Plantilla de clase del adaptador queue de la Biblioteca estándar.
3  #include <iostream>
4  #include <queue> // definición del adaptador queue
5  using namespace std;
6
7  int main()
8  {
9     queue< double > valores; // cola con valores double
10
11    // mete los elementos en la cola valores
12    valores.push( 3.2 );
13    valores.push( 9.8 );
14    valores.push( 5.4 );
15
16    cout << "Sacando datos de valores: ";
17
18    // saca los elementos de la cola
19    while ( !valores.empty() )
20    {
21       cout << valores.front() << ' '; // ve el elemento que está al frente
22       valores.pop(); // elimina el elemento
23    } // fin de while
24
25    cout << endl;
26 } // fin de main
```

```
Sacando datos de valores: 3.2 9.8 5.4
```

Fig. 15.20 | Plantillas de clase del adaptador queue de la Biblioteca estándar.

15.7.3 Adaptador priority_queue

La clase **priority_queue** (del encabezado <queue>) ofrece una funcionalidad que permite *inserciones ordenadas* en la estructura de datos subyacente, y eliminaciones de su *parte inicial*. De manera predeterminada, los elementos de un priority_queue se almacenan en un vector. Al agregar elementos a un adaptador priority_queue, éstos se insertan en *orden de prioridad* de forma que el elemento con la prioridad más alta (es decir, el valor *más grande*) sea el primer elemento removido del adaptador priority_queue. Esto se logra generalmente al disponer los elementos en una estructura de datos conocida como **heap** o montón (no debe confundirse con el montón para la memoria asignada en forma dinámica) que siempre mantiene el valor más grande (es decir, el de mayor prioridad) en la parte inicial de la estructura de datos. En la sección 16.3.12 usaremos los *algoritmos de heap* de la Biblioteca estándar. La comparación de elementos se lleva a cabo con el *objeto función de comparación* less<T> de manera predeterminada, pero el programador puede suministrar un comparador distinto.

Hay varias operaciones comunes de un adaptador priority_queue. La función push inserta un elemento en la posición apropiada, con base en el *orden de prioridad* del adaptador priority_queue (lo que se implementa mediante una llamada a la función push_back del contenedor subyacente y luego se vuelven a ordenar los elementos por prioridad). La función pop elimina el elemento de *mayor prioridad* en el adaptador priority_queue (lo que se implementa mediante una llamada a la función pop_back del contenedor subyacente, después de eliminar el elemento superior del montón). **top** obtiene una

referencia al elemento *superior* del adaptador `priority_queue` (lo que se implementa mediante una llamada a la función `front` del contenedor subyacente). `empty` determina si el adaptador `priority_queue` está *vacío* o no (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente). `size` obtiene el número de elementos en el adaptador (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente).

La figura 15.21 demuestra el uso de la clase de adaptador `priority_queue`. En la línea 9 se crea una instancia de `priority_queue` que almacena valores `double` y utiliza un `vector` como la estructura de datos subyacente. En las líneas 12 a 14 se utiliza la función `push` para agregar elementos al adaptador `priority_queue`. La instrucción `while` en las líneas 19 a 23 utiliza la función `empty` (disponible en *todos* los contenedores) para determinar si el adaptador está vacío o no (línea 19). Mientras haya más elementos, en la línea 21 se utiliza la función `top` de `priority_queue` para obtener el elemento de *mayor prioridad* (es decir, el de mayor valor) en el adaptador y mostrarlo en pantalla. En la línea 22 se elimina el elemento de *mayor prioridad* del adaptador mediante la función `pop` (disponible en todas las clases de adaptadores).

```
1  // Fig. 15.21: fig15_21.cpp
2  // Clase del adaptador priority_queue de la Biblioteca estándar.
3  #include <iostream>
4  #include <queue> // definición del adaptador priority_queue
5  using namespace std;
6
7  int main()
8  {
9     priority_queue< double > prioridades; // crea un priority_queue
10
11    // mete elementos en prioridades
12    prioridades.push( 3.2 );
13    prioridades.push( 9.8 );
14    prioridades.push( 5.4 );
15
16    cout << "Sacando datos de prioridades: ";
17
18    // saca elemento de priority_queue
19    while ( !prioridades.empty() )
20    {
21       cout << prioridades.top() << ' '; // ve el elemento superior
22       prioridades.pop(); // elimina el elemento superior
23    } // fin de while
24
25    cout << endl;
26 } // fin de main
```

```
Sacando datos de prioridades: 9.8 5.4 3.2
```

Fig. 15.21 | Clase de adaptador `priority_queue` de la Biblioteca estándar.

15.8 La clase `bitset`

La clase **`bitset`** facilita la creación y manipulación de **conjuntos de bits**, los cuales son útiles para representar un conjunto de banderas de bits. Los objetos `bitset` tienen un tamaño fijo en tiempo de compilación. La clase `bitset` es una herramienta alternativa para la *manipulación de bits*, que vimos en el capítulo 22.

La declaración

```
bitset< tamanio > b;
```

crea el objeto `bitset` `b`, en el que todos los bits son inicialmente `0` (“apagado”).

La instrucción

```
b.set( numeroBit );
```

hace que el bit `numeroBit` del objeto `bitset` `b` se “encienda”. La expresión `b.set()` hace que todos los bits en `b` se “enciendan”.

La instrucción

```
b.reset( numeroBit );
```

hace que el bit `numeroBit` del objeto `bitset` `b` se “apague”. La expresión `b.reset()` hace que todos los bits en `b` se “apaguen”.

La instrucción

```
b.flip( numeroBit );
```

“voltea” el bit `numeroBit` del objeto `bitset` `b` (es decir, si el bit está “encendido”, `flip` lo “apaga”). La expresión `b.flip()` voltea todos los bits en `b`.

La instrucción

```
b[ numeroBit ];
```

devuelve una referencia al bit `numeroBit` del objeto `bitset` `b`. De manera similar,

```
b.at( numeroBit );
```

realiza primero una comprobación de rango en `numeroBit`. Después, si `numeroBit` se encuentra dentro del rango, `at` devuelve una referencia a ese bit. En caso contrario, `at` lanza una excepción `out_of_range`.

La instrucción

```
b.test( numeroBit );
```

realiza primero una *comprobación de rango* en `numeroBit`. Después, si `numeroBit` se encuentra dentro del rango, `test` devuelve `true` si el bit está encendido y `false` si está apagado. En caso contrario, `test` lanza una excepción `out_of_range`.

La expresión

```
b.size()
```

devuelve el número de bits en el objeto `bitset` `b`.

La expresión

```
b.count()
```

devuelve el número de bits encendidos en el objeto `bitset` `b`.

La expresión

```
b.any()
```

devuelve `true` si alguno de los bits en el objeto `bitset` `b` está encendido.

La expresión

```
b.all()
```

devuelve `true` si todos los bits en el objeto `bitset` `b` están encendidos.

La expresión

```
b.none()
```

devuelve `true` si ninguno de los bits en el objeto `bitset b` está encendido.

Las expresiones

```
b == b1
b != b1
```

comparan los dos objetos `bitset` para ver si son iguales y desiguales, respectivamente.

Cada uno de los operadores de asignación a nivel de bits `&=`, `|=` y `^=` (que veremos con detalle en la sección 22.5) puede utilizarse para combinar objetos `bitset`. Por ejemplo,

```
b &= b1;
```

realiza una operación AND lógica bit por bit, entre los objetos `bitset b` y `b1`. El resultado se almacena en `b`. Las operaciones OR y XOR lógicas a nivel de bits se realizan mediante

```
b |= b1;
b ^= b2;
```

La expresión

```
b >>= n;
```

desplaza los bits en el objeto `bitset b` a la derecha, `n` posiciones.

La expresión

```
b <<= n;
```

desplaza los bits en el objeto `bitset b` a la izquierda, `n` posiciones.

Las expresiones

```
b.to_string()
b.to_ulong()
```

convierten el objeto `bitset b` en un objeto `string` y en un `unsigned long`, respectivamente.

15.9 Conclusión

En este capítulo presentamos tres componentes clave de la Biblioteca estándar: contenedores, iteradores y algoritmos. Usted aprendió acerca de los *contenedores de secuencia* lineales: `array` (capítulo 7), `vector`, `deque`, `forward_list` y `list`, que representan estructuras de datos lineales. Hablamos sobre los *contenedores asociativos* no lineales `set`, `multiset`, `map` y `multimap`, además de sus versiones desordenadas. También vimos que los *adaptadores de contenedores* `stack`, `queue` y `priority_queue` pueden utilizarse para restringir las operaciones de los contenedores de secuencia `vector`, `deque` y `list` para el propósito de implementar las estructuras de datos especializadas representadas por los adaptadores de contenedores. Aprendió acerca de las categorías de iteradores y que cada algoritmo se puede utilizar con cualquier contenedor que soporte la mínima funcionalidad de iteradores que requiere el algoritmo. Aprendió además acerca de la clase `bitset`, que facilita la creación y manipulación de conjuntos de bits como un contenedor.

En el siguiente capítulo continuaremos nuestra discusión sobre los contenedores, iteradores y algoritmos de la Biblioteca estándar con un análisis detallado de los algoritmos. También aprenderá acerca de los apuntadores de funciones, objetos de funciones y las nuevas expresiones lambda de C++11.

Resumen

Sección 15.1 Introducción

- La Biblioteca estándar de C++ define componentes reutilizables poderosos y basados en plantillas para estructuras de datos comunes, además de algoritmos que se utilizan para procesar esas estructuras de datos.
- Hay tres categorías de clases de contenedores: contenedores de primera clase, adaptadores de contenedores y casi contenedores.
- Los iteradores, que tienen propiedades similares a las de los apuntadores, se utilizan para manipular elementos de los contenedores.
- Los algoritmos de la Biblioteca estándar son plantillas de funciones que realizan manipulaciones de datos comunes, tales como búsqueda, ordenamiento y comparación de elementos o de contenedores completos.

Sección 15.2 Introducción a los contenedores

- Los contenedores se dividen en contenedores de secuencia, contenedores asociativos ordenados, contenedores asociativos desordenados y adaptadores de contenedores (pág. 640).
- Los contenedores de secuencia (pág. 640) representan estructuras de datos lineales.
- Los contenedores asociativos son contenedores no lineales que localizan con rapidez los elementos almacenados en ellos, como conjuntos de valores o pares clave/valor (pág. 641).
- Los contenedores de secuencia y los contenedores asociativos se conocen en forma colectiva como contenedores de primera clase.
- Las plantillas de clase `stack`, `queue` y `priority_queue` son adaptadores de contenedores que permiten a un programa ver un contenedor de secuencia de una manera limitada.
- Los casi contenedores (pág. 641; arreglos integrados, `bitset` y `valarray`) exhiben capacidades similares a las de los contenedores de primera clase, pero no soportan todas las capacidades de los contenedores de primera clase.
- La mayoría de los contenedores cuentan con una funcionalidad similar. Muchas operaciones se aplican a todos los contenedores y otras operaciones se aplican a subconjuntos de contenedores similares.
- Los contenedores de primera clase definen muchos tipos anidados comunes que se utilizan en declaraciones de variables basadas en plantillas, parámetros a funciones y valores de retorno de funciones.

Sección 15.3 Introducción a los iteradores

- Los iteradores tienen muchas similitudes con los apuntadores y se utilizan para apuntar a elementos de contenedores de primera clase.
- La función `begin` (pág. 644) de los contenedores de primera clase devuelve un iterador que apunta al primer elemento de un contenedor. La función `end` (pág. 644) devuelve un iterador que apunta después del último elemento del contenedor (un elemento más allá del final); generalmente se utiliza en un ciclo para indicar cuándo se debe terminar el procesamiento de los elementos del contenedor.
- Un `istream_iterator` (pág. 645) es capaz de extraer los valores de un flujo de entrada en forma segura para los tipos. Un `ostream_iterator` (pág. 645) es capaz de insertar valores en un flujo de salida.
- Un iterador de acceso aleatorio (pág. 647) tiene las herramientas de un iterador bidireccional y la habilidad de acceder directamente a cualquier elemento del contenedor.
- Un iterador bidireccional (pág. 646) tiene las herramientas de un iterador de avance y la habilidad de desplazarse en dirección hacia atrás.
- Un iterador de avance (pág. 646) combina las herramientas de los iteradores de entrada y salida.
- Los iteradores de entrada y salida (pág. 646) sólo pueden desplazarse en dirección de avance, un elemento a la vez.

Sección 15.4 Introducción a los algoritmos

- Los algoritmos de la Biblioteca estándar operan sobre elementos de contenedores sólo en forma indirecta, a través de los iteradores.
- Muchos algoritmos operan sobre secuencias de elementos definidas por iteradores que apuntan al primer elemento de la secuencia y a una posición más allá del último elemento.

Sección 15.5 Contenedores de secuencia

- La Biblioteca estándar proporciona los contenedores de secuencia `array`, `vector`, `forward_list`, `list` y `deque`. Las plantillas de clases `array`, `vector` y `deque` están basadas en arreglos integrados. Las plantillas de clase `forward_list` y `list` implementan una estructura de datos tipo lista enlazada.

Sección 15.5.1 Contenedor de secuencia `vector`

- La función `capacity` (pág. 652) devuelve el número de elementos que se pueden almacenar en un `vector` antes de que éste cambie su tamaño en forma dinámica, para dar cabida a más elementos.
- La función `push_back` (pág. 652) de los contenedores de secuencia agrega un elemento al final de un contenedor.
- La función miembro `cbegin` (pág. 653; nueva en C++11) de `vector` devuelve un `const_iterator` al primer elemento del `vector`.
- La función miembro `cend` (pág. 653; nueva en C++11) de `vector` devuelve un `const_iterator` a la ubicación más allá del último elemento del `vector`.
- La función miembro `crbegin` (pág. 654; nueva en C++11) devuelve un `const_reverse_iterator` al último elemento del `vector`.
- La función miembro `crend` (pág. 654; nueva en C++11) devuelve un `const_reverse_iterator` a la ubicación antes del primer elemento del `vector`.
- A partir de C++11, es posible pedir a un `vector` o `deque` que devuelva la memoria que no necesita al sistema, mediante una llamada a la función miembro `shrink_to_fit` (pág. 654).
- A partir de C++11, es posible usar inicializadores de lista para inicializar los elementos de `vector` y otros tipos de contenedores.
- El algoritmo `copy` (pág. 656; del encabezado `<algorithm>`) copia cada elemento en un rango, empezando con la ubicación especificada por el iterador en su primer argumento, y hasta (pero sin incluir) la ubicación especificada por el iterador en su segundo argumento.
- La función `front` (pág. 656) devuelve una referencia al primer elemento en un contenedor de secuencia. La función `begin` devuelve un iterador que apunta al principio de un contenedor de secuencia.
- La función `back` (pág. 656) devuelve una referencia al último elemento en un contenedor de secuencia (excepto `forward_list`). La función `end` devuelve un iterador que apunta a un elemento más allá del final de un contenedor de secuencia.
- La función `insert` (pág. 657) de los contenedores de secuencia inserta uno o varios valores antes del elemento en una ubicación específica y devuelve un iterador que apunta al elemento insertado o al primero de los elementos insertados.
- La función `erase` (pág. 657; en todos los contenedores de primera clase excepto `forward_list`) elimina uno o varios elementos específicos del contenedor.
- La función `empty` (pág. 657; en todos los contenedores y adaptadores) devuelve `true` si el contenedor está vacío.
- La función `clear` (pág. 658; en todos los contenedores de primera clase) vacía el contenedor.

Sección 15.5.2 Contenedor de secuencia `list`

- El contenedor de secuencia `list` (pág. 658; del encabezado `<list>`) implementa una lista con enlace doble que proporciona una implementación eficiente para las operaciones de inserción y eliminación en cualquier ubicación del contenedor.
- El contenedor de secuencia `forward_list` (pág. 658; del encabezado `<forward_list>`) implementa una lista con enlace simple que sólo soporta iteradores de avance.
- La función miembro `push_front` (pág. 661) de `list` inserta valores al principio de una lista.
- La función miembro `sort` (pág. 661) de `list` sortea los elementos de la lista en orden ascendente.
- La función miembro `splice` (pág. 661) de `list` elimina elementos en un objeto `list` y los inserta en otro objeto `list` en una posición específica.
- La función miembro `unique` (pág. 662) de `list` elimina los elementos duplicados en un objeto `list`.

- La función miembro `assign` (pág. 662) de `list` reemplaza el contenido de un objeto `list` con el contenido de otro.
- La función miembro `remove` (pág. 662) de `list` elimina todas las copias de un valor especificado de un objeto `list`.

Sección 15.5.3 Contenedor de secuencia `deque`

- La plantilla de clase `deque` (pág. 663) proporciona las mismas operaciones que un objeto `vector`, pero agrega las funciones miembro `push_front` y `pop_front` (pág. 662) para permitir la inserción y eliminación de elementos al principio de un objeto `deque`, respectivamente. Debe incluirse el encabezado `<deque>` para utilizar la plantilla de clase `deque`.

Sección 15.6 Contenedores asociativos

- Los contenedores asociativos de la Biblioteca estándar proporcionan acceso directo para almacenar y obtener elementos a través de claves (pág. 664).
- Los cuatro contenedores asociativos ordenados (pág. 664) son `multiset`, `set`, `multimap` y `map`.
- Los cuatro contenedores asociativos desordenados (pág. 664) son `unordered_multiset`, `unordered_set`, `unordered_multimap` y `unordered_map`. Éstos son casi idénticos a sus contrapartes ordenados, pero no mantienen las claves en orden.
- Las plantillas de clases `multiset` y `set` proporcionan operaciones para manipular conjuntos de valores, en donde los valores son las claves; no hay un valor separado asociado con cada clave. Debe incluirse el encabezado `<set>` para utilizar las plantillas de clases `set` y `multiset`.
- Un `multiset` permite claves duplicadas y un `set` no.

Sección 15.6.1 Contenedor asociativo `multiset`

- El contenedor asociativo `multiset` (pág. 664) proporciona operaciones rápidas de almacenamiento y obtención de claves, y permite claves duplicadas. El orden de los elementos se determina mediante un objeto de función de comparación. Si el orden de las claves no es importante, podemos usar `unordered_multiset` (encabezado `<unordered_set>`) como alternativa.
- Las claves de un objeto `multiset` se pueden ordenar en forma ascendente, para lo cual se ordenan las claves con el objeto de función de comparación `less<T>` (pág. 665).
- El tipo de las claves en todos los contenedores asociativos debe soportar la comparación en forma apropiada, con base en el objeto de función de comparación especificado.
- Un objeto `multiset` soporta los iteradores bidireccionales.
- Debe incluirse el encabezado `<set>` (pág. 665) para utilizar la clase `multiset`.
- La función `count` (pág. 667; disponible para todos los contenedores asociativos) cuenta el numero de ocurrencias del valor especificado que se encuentra actualmente en un contenedor.
- La función `find` (pág. 667; disponible para todos los contenedores asociativos) localiza un valor especificado en un contenedor.
- Las funciones `lower_bound` y `upper_bound` (pág. 667) de los contenedores asociativos localizan la primera ocurrencia del valor especificado en un contenedor y el elemento después de la última ocurrencia del valor, respectivamente.
- La función `equal_range` (pág. 667) de los contenedores asociativos devuelve un `pair` que contiene los resultados de las operaciones `lower_bound` y `upper_bound`.
- C++ también incluye la plantilla de clase `tuple`, que es similar a `pair` pero puede contener cualquier número de elementos de diversos tipos.

Sección 15.6.2 Contenedor asociativo `set`

- El contenedor asociativo `set` se utiliza para operaciones rápidas de almacenamiento y obtención de claves únicas. Si el orden de las claves no es importante, podemos usar `unordered_set` (encabezado `<unordered_set>`) como alternativa.
- Si se hace un intento de insertar una clave duplicada en un objeto `set`, se ignora el duplicado.

- Un objeto `set` soporta los iteradores bidireccionales.
- Debe incluirse el encabezado `<set>` para utilizar la clase `set`.

Sección 15.6.3 Contenedor asociativo `multimap`

- Los contenedores `multimap` y `map` proporcionan operaciones para manipular los pares clave-valor. Si el orden de las claves no es importante, podemos usar `unordered_multimap` (encabezado `unordered_map`) como alternativa.
- La principal diferencia entre un objeto `multimap` y un objeto `map` es que un `multimap` permite almacenar claves duplicadas con valores asociados, y un `map` sólo permite claves únicas con valores asociados.
- El contenedor asociativo `multimap` se utiliza para almacenar y recuperar con rapidez los pares clave-valor.
- Las claves duplicadas se permiten en un `multimap`, por lo que es posible asociar múltiples valores con una sola clave. A esto se le conoce como relación de uno a varios.
- El encabezado `map` (pág. 669) debe incluirse para usar las plantillas de clases `map` y `multimap`.
- La función `make_pair` crea de manera automática un `pair` mediante el uso de los tipos especificados en la declaración de `multimap`.
- En C++11, si conocemos los pares clave-valor de antemano, podemos usar la inicialización con listas al crear un `multimap`.

Sección 15.6.4 Contenedor asociativo `map`

- No se permiten claves duplicadas en un objeto `map`, por lo que sólo se puede asociar un solo valor con cada clave. A esto se le conoce como asignación de uno a uno (pág. 671). Si no es importante el orden de las claves, podemos usar `unordered_map` (encabezado `<unordered_map>`) como alternativa.

Sección 15.7 Adaptadores de contenedores

- Los adaptadores de contenedores son: `stack`, `queue` y `priority_queue`.
- Los adaptadores no son contenedores de primera clase, ya que no proporcionan la implementación de la estructura de datos actual en la que se pueden almacenar elementos, y no soportan iteradores.
- Las tres plantillas de clases de adaptadores proporcionan las funciones miembro `push` y `pop` (pág. 673) que insertan de manera apropiada un elemento en, y lo eliminan de, cada estructura de datos del adaptador, respectivamente.

Sección 15.7.1 Adaptador `stack`

- La plantilla de clase `stack` (pág. 673) es una estructura de datos del tipo "último en entrar, primero en salir". Debe incluirse el encabezado `<stack>` (pág. 673) para utilizar la plantilla de clase `stack`.
- La función miembro `top` (pág. 673) de `stack` devuelve una referencia al elemento superior del objeto `stack` (lo que se implementa mediante una llamada a la función `back` del contenedor subyacente).
- La función miembro `empty` de `stack` determina si el objeto `stack` está vacío (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente).
- La función miembro `size` de `stack` devuelve el número de elementos en el objeto `stack` (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente).

Sección 15.7.2 Adaptador `queue`

- La plantilla de clase `queue` (pág. 675) implementa una estructura de datos PEPS. Debe incluirse el encabezado `<queue>` (pág. 675) para utilizar un objeto `queue` o un `priority_queue`.
- La función miembro `front` de `queue` devuelve una referencia al primer elemento en el objeto `queue`.
- La función miembro `back` (pág. 675) de `queue` devuelve una referencia al último elemento en el objeto `queue`.
- La función miembro `empty` de `queue` determina si el objeto `queue` está vacío.
- La función miembro `size` de `queue` devuelve el número de elementos en el objeto `queue`.

Sección 15.7.3 Adaptador `priority_queue`

- La plantilla de clase `priority_queue` proporciona una funcionalidad que permite inserciones ordenadas en la estructura de datos subyacente, y eliminaciones de la parte frontal de la estructura de datos subyacente.
- Las operaciones comunes de `priority_queue` (pág. 676) son `push`, `pop`, `top`, `empty` y `size`.

Sección 15.8 La clase `bitset`

- La plantilla de clase `bitset` (pág. 677) facilita la creación y manipulación de conjuntos de bits, que son útiles para representar un conjunto de banderas de bits.

Ejercicios de autoevaluación

15.1 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) El código basado en apuntador es complejo y propenso a errores; las más ligeras omisiones o descuidos pueden provocar graves violaciones de acceso a la memoria y errores de fuga de memoria por los que el compilador emitirá una advertencia.
b) Los `deques` ofrecen operaciones rápidas de inserción y eliminación en la parte frontal o trasera, además de acceso directo a cualquier elemento.
c) Los contenedores `list` son listas con enlace simple y ofrecen operaciones rápidas de inserción y eliminación en cualquier parte.
d) Los contenedores `multimap` ofrecen la asignación de uno a varios en donde se permiten duplicados, y una búsqueda rápida basada en claves.
e) Los contenedores asociativos son estructuras de datos no lineales que por lo general pueden localizar los elementos almacenados en los contenedores con rapidez.
f) La función miembro de contenedor `cbegin` devuelve un `iterator` que hace referencia al primer elemento del contenedor.
g) La operación `++` en un iterador lo desplaza al siguiente elemento del contenedor.
h) Cuando se aplica el operador `*` (desreferencia) a un iterador `const`, se devuelve una referencia `const` al elemento contenedor, lo que permite el uso de funciones miembro que no sean `const`.
i) El uso de iteradores en donde sea apropiado es otro ejemplo del principio del menor privilegio.
j) Muchos algoritmos operan sobre secuencias de elementos definidas por iteradores que apuntan al primer elemento de la secuencia y al último elemento.
k) La función `capacity` devuelve el número de elementos que pueden almacenarse en el `vector` antes de que éste necesite cambiar su tamaño en forma dinámica para dar cabida a más elementos.
l) Uno de los usos más comunes de un `deque` es para mantener una cola de elementos del tipo "primero en entrar, primero en salir". De hecho, un `deque` es la implementación subyacente predeterminada para el adaptador tipo cola.
m) `push_front` está disponible sólo para la clase `list`.
n) Sólo pueden realizarse inserciones y eliminaciones en la parte frontal y trasera de un `map`.
o) La clase `queue` permite inserciones en la parte frontal de la estructura de datos subyacente y eliminaciones de la parte trasera (lo que se conoce comúnmente como una estructura de datos del tipo "primero en entrar, primero en salir").

15.2 Llene los espacios en blanco en cada uno de los siguientes enunciados:

a) Los tres componentes clave de la porción correspondiente a la "STL" de la Biblioteca estándar son: _____, _____ y _____.
b) Los arreglos integrados pueden manipularse mediante algoritmos de la Biblioteca estándar, usando _____ como iteradores.
c) El adaptador de contenedor de la Biblioteca estándar más estrechamente asociado con la disciplina de inserción y eliminación del tipo "último en entrar, primero en salir (UEPS)" es _____.
d) Los contenedores de secuencia y los contenedores _____ se conocen en forma colectiva como contenedores de primera clase.
e) Un constructor _____ inicializa el contenedor para que sea una copia de un contenedor existente del mismo tipo.

f) La función miembro de contenedor _____ devuelve `true` si no hay elementos en el contenedor; de lo contrario devuelve `false`.
g) La función miembro de contenedor _____ (nueva en C++11) desplaza los elementos de un contenedor a otro; esto evita la sobrecarga de copiar cada elemento del contenedor que se incluye como argumento.
h) La función miembro de contenedor _____ se sobrecarga para devolver un `iterator` o un `const_iterator` que hace referencia al primer elemento del contenedor.
i) Las operaciones realizadas en un `const_iterator` devuelven _____ para evitar la modificación de los elementos del contenedor que se está manipulando.
j) Los contenedores de secuencia de la Biblioteca estándar son `array`, `vector`, `deque`, _____ y _____.
k) Seleccione el contenedor _____ para el mejor rendimiento de acceso aleatorio en un contenedor que pueda aumentar de tamaño.
l) La función `push_back`, que está disponible en los contenedores de secuencia distintos de _____, agrega un elemento al final del contenedor.
m) Al igual que con `cbegin` y `cend`, ahora C++11 incluye las funciones miembro `crbegin` y `crend` de `vector`, que devuelven _____ que representan los puntos inicial y final al iterar a través de un contenedor en sentido inverso.
n) Una función _____ unaria recibe un solo argumento, realiza una comparación usando ese argumento y devuelve un valor `bool` que indica el resultado.
o) La principal diferencia entre los contenedores asociativos ordenados y desordenados es _____.
p) La principal diferencia entre un `multimap` y un `map` es _____.
q) C++11 introduce la plantilla de clase `tuple`, que es similar a `pair` pero puede _____.
r) El contenedor asociativo `map` realiza operaciones rápidas de almacenamiento y recuperación de claves únicas y sus valores asociados. No se permiten claves duplicadas; puede asociarse un solo valor a cada clave. A esto se le conoce como asignación _____.
s) La clase _____ cuenta con funcionalidad que permite inserciones ordenadas en la estructura de datos subyacente y eliminaciones de la parte frontal de la estructura de datos subyacente.

15.3 Escriba una instrucción o expresión que realice cada una de las siguientes tareas de `bitset`:
a) Escriba una declaración que cree `banderas` tipo `bitset` de tamaño `tamanio`, en donde cada bit sea inicialmente 0.
b) Escriba una instrucción que "apague" el bit `numeroBit` del `bitset banderas`.
c) Escriba una instrucción que devuelva una referencia al bit `numeroBit` del `bitset banderas`.
d) Escriba una expresión que devuelva el número de bits que están encendidos en el `bitset banderas`.
e) Escriba una expresión que devuelva `true` si están encendidos todos los bits en el `bitset banderas`.
f) Escriba una expresión que compare el `bitset banderas` con `otrasBanderas` para ver si son desiguales.
g) Escriba una expresión que desplace los bits en el `bitset banderas` por `n` posiciones a la izquierda.

Respuestas a los ejercicios de autoevaluación

15.1 a) Falso. El compilador no advierte sobre estos tipos de errores en tiempo de ejecución. b) Verdadero. c) Falso. Son listas con doble enlace. d) Verdadero. e) Verdadero. f) Falso. Devuelve un `const_iterator`. g) Verdadero. h) Falso. No permite el uso de funciones miembro que no sean `const`. i) Falso. El uso de `const_iterators` en donde sea apropiado es otro ejemplo del principio del menor privilegio. j) Falso. Muchos algoritmos operan sobre secuencias de elementos definidos por iteradores que apuntan al primer elemento de la secuencia y a un elemento más allá del último elemento. k) Verdadero. l) Verdadero. m) Falso. También está disponible para la clase `deque`. n) Falso. Pueden hacerse inserciones y eliminaciones en cualquier parte de un `map`. o) Falso. Pueden ocurrir inserciones sólo en la parte trasera y pueden ocurrir eliminaciones sólo en la parte frontal.

15.2 a) contenedores, iteradores y algoritmos. b) apuntadores. c) `stack`. d) asociativos. e) de copia. f) `empty`. g) versión de movimiento de `operator=`. h) `begin`. i) Referencias `const`. j) `list` y `forward_list`. k) `vector`. l) `array`. m) iteradores `const_reverse_iterator`. n) predicado. o) los desordenados no mantienen sus claves ordenadas. p) Un `multimap` permite almacenar claves duplicadas con sus valores asociados y un `map` sólo permite

claves únicas con sus valores asociados. q) contener cualquier cantidad de elementos de diversos tipos. r) uno a uno. s) `priority_queue`.

15.3 a) `bitset< size > banderas;`
b) `banderas.reset( numeroBit );`
c) `banderas[ numeroBit ];`
d) `banderas.count()`
e) `banderas.all()`
f) `banderas != otrasBanderas`
g) `banderas <<= n;`

Ejercicios

15.4 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Muchos de los algoritmos de la Biblioteca estándar pueden aplicarse a diversos contenedores, independientemente de la implementador del contenedor subyacente.
b) Los contenedores `array` son de tamaño fijo y ofrecen acceso directo a cualquier elemento.
c) Los contenedores `forward_list` son listas con enlace simple, que ofrecen operaciones rápidas de inserción y eliminación sólo en la parte frontal y en la parte trasera.
d) Los contenedores `set` ofrecen búsqueda rápida y se permiten duplicados.
e) En un contenedor `priority_queue`, el elemento de menor prioridad siempre es el primer elemento que se saca.
f) Los contenedores de secuencia representan estructuras de datos no lineales.
g) A partir de C++11, ahora hay una versión de función no miembro de `swap` que intercambia el contenido de sus dos argumentos (que deben ser de distintos tipos de contenedores) mediante el uso de operaciones de movimiento, en vez de operaciones de copia.
h) La función miembro de contenedor `erase` elimina todos los elementos del contenedor.
i) Un objeto de tipo `iterator` hace referencia a un elemento de un contenedor que *puede* modificarse.
j) Utilizamos versiones `const` de los iteradores para recorrer contenedores de sólo lectura.
k) Para los iteradores de entrada y de salida, es común guardar el iterador y después usar el valor guardado más adelante.
l) Las plantillas de clases `array`, `vector` y `deque` se basan en arreglos integrados.
m) Tratar de desreferenciar un iterador colocado fuera de su contenedor es un error de compilación. En particular, el iterador devuelto por `end` no debe desreferenciarse o incrementarse.
n) Las inserciones y eliminaciones en medio de un contenedor `deque` están optimizadas para minimizar el número de elementos copiados, por lo que es más eficiente que un `vector` pero menos eficiente que un contenedor `list` para este tipo de modificación.
o) El contenedor `set` *no* permite duplicados.
p) La clase `stack` (del encabezado `<stack>`) permite inserciones en, y eliminaciones de, la estructura de datos subyacente en un extremo (lo que se conoce comúnmente como estructura de datos tipo "primero en entrar, primero en salir").
q) La función `empty` está disponible en todos los contenedores, excepto `deque`.

15.5 Llene los espacios en blanco en cada uno de los siguientes enunciados:

a) Los tres estilos de clases de contenedores son: contenedores de primera clase, _____ y casi contenedores.
b) Los contenedores se dividen en cuatro categorías principales: contenedores de secuencia, contenedores asociativos ordenados, _____ y adaptadores de contenedores.
c) El adaptador de contenedor de la Biblioteca estándar más estrechamente asociado con la disciplina de inserción y eliminación del tipo "primero en entrar, primero en salir (UEPS)" es _____.
d) Los arreglos integrados, los `bitset` y los `valarray` son todos contenedores _____.
e) Un constructor _____ (nuevo en C++11) desplaza el contenido de un contenedor existente del mismo tipo a un nuevo contenedor, sin la sobrecarga de copiar cada elemento del contenedor que se incluye como argumento.

f) La función miembro de contenedor _____ devuelve el número de elementos actualmente en el contenedor.
g) La función miembro de contenedor _____ devuelve `true` si el contenido del primer contenedor no es igual al contenido del segundo; en caso contrario devuelve `false`.
h) Usamos iteradores con secuencias; éstas pueden ser secuencias de entrada o de salida, o pueden ser _____.
i) Los algoritmos de la Biblioteca estándar operan sobre elementos contenedores sólo de manera indirecta, a través de _____.
j) Las aplicaciones con inserciones y eliminaciones frecuentes en la parte media y/o en los extremos de un contenedor utilizan por lo general un(a) _____.
k) La función _____ está disponible en *todos* los contenedores de primera clase (excepto `forward_list`) y devuelve el número de elementos almacenados actualmente en el contenedor.
l) Puede ser un desperdicio duplicar el tamaño de un `vector` cuando se necesita más espacio. Por ejemplo, un `vector` lleno de 1 000 000 elementos ajusta su tamaño para dar cabida a 2 000 000 elementos cuando se agrega un nuevo elemento, dejando 999 999 elementos sin usar. Es posible usar _____ y _____ para controlar mejor el uso del espacio.
m) A partir de C++11, podemos pedir a un `vector` o `deque` que devuelva la memoria innecesaria al sistema mediante una llamada a la función miembro _____.
n) Los contenedores asociativos proporcionan acceso directo para almacenar y recuperar elementos mediante claves (lo que se conoce comúnmente como claves de búsqueda). Los contenedores asociativos ordenados son `multiset`, `set`, _____ y _____.
o) Las clases _____ y _____ proporcionan operaciones para manipular conjuntos de valores en donde éstos son las claves; *no* hay un valor separado asociado con cada clave.
p) Usamos la palabra clave `auto` de C++11 _____.
q) Un `multimap` se implementa para localizar de manera eficiente todos los valores que hacen pares con una _____ dada.
r) Los adaptadores de contenedores de la Biblioteca estándar son `stack`, `queue` y _____.

Preguntas de discusión

15.6 Explique por qué el hecho de usar el "iterador más débil" que produce un rendimiento aceptable ayuda a producir componentes que pueden reutilizarse al máximo.

15.7 ¿Por qué es costoso insertar (o eliminar) un elemento en medio de un `vector`?

15.8 Los contenedores que soportan iteradores de acceso aleatorio pueden usarse con la mayoría, pero no con todos los algoritmos de la Biblioteca estándar. ¿Cuál es la excepción?

15.9 ¿Por qué utilizaría el operador * para desreferenciar un iterador?

15.10 ¿Por qué es eficiente la inserción en la parte trasera de un `vector`?

15.11 ¿Cuándo es preferible usar un `deque` en vez de un `vector`?

15.12 Describa lo que ocurre al tratar de insertar un elemento en un vector cuya memoria esté agotada.

15.13 ¿Cuándo preferiría un contenedor `list` en vez de un `deque`?

15.14 ¿Qué ocurre cuando el subíndice del contenedor `map` es una clave que no está en el mapa?

15.15 Use inicializadores de lista de C++11 para inicializar el `vector nombres` con las cadenas `" Suzanne"`, `"James"`, `"Maria"` y `"Juan"`. Muestre ambas sintaxis comunes.

15.16 ¿Qué ocurre cuando eliminamos un elemento de un contenedor que contiene un apuntador a un objeto asignado en forma dinámica?

15.17 Describa el contenedor asociativo ordenado `multiset`.

15.18 ¿Cómo podría usarse un contenedor asociativo ordenado `multimap` en un sistema de procesamiento de transacciones de tarjetas de crédito?

15.19 Escriba una instrucción que cree e inicialice un `multimap` de objetos `string` y valores `int` con tres pares clave-valor.

15.20 Explique las operaciones `push`, `pop` y `top` de un contenedor `stack`.

15.21 Explique las operaciones `push`, `pop`, `front` y `back` de un contenedor `queue`.

15.22 ¿Qué diferencia hay entre insertar un elemento en un contenedor `priority_queue` e insertar un elemento en casi cualquier otro contenedor?

Ejercicios de programación

15.23 ***(Palíndromos)*** Escriba una plantilla de función llamada `palindromo` que reciba un parámetro `vector` y devuelva `true` o `false`, dependiendo de si el `vector` se lee igual o no tanto hacia adelante como hacia atrás (por ejemplo, un `vector` que contenga 1, 2, 3, 2, 1 es un palíndromo, pero un `vector` que contenga 1, 2, 3, 4 no lo es).

15.24 ***(Criba de Eratóstenes con `bitset`)*** Este ejercicio reformula la *Criba de Eratóstenes* para encontrar números primos que vimos en el ejercicio 7.27. Use un contenedor `bitset` para implementar el algoritmo. Su programa debe mostrar todos los números primos del 2 al 1023, para después permitir al usuario que introduzca un número para determinar si es primo o no.

15.25 ***(Criba de Eratóstenes)*** Modifique el ejercicio 15.24, la Criba de Eratóstenes, de manera que, si el número que introduce el usuario en el programa no es primo, éste muestre los factores primos del número. Recuerde que los factores de un número primo son sólo 1 y el número primo en sí. Todo número no primo tiene una factorización prima única. Por ejemplo, los factores del número 54 son 2, 3, 3 y 3. Cuando se multiplican esos valores, el resultado es 54. Para el número 54, la salida de factores primos debe ser 2 y 3.

15.26 ***(Factores primos)*** Modifique el ejercicio 15.25 de modo que, si el número que introduce el usuario en el programa no es primo, éste debe mostrar los factores primos del número además de la cantidad de veces que aparezca cada factor primo en la factorización prima única. Por ejemplo, la salida para el número 54 debería ser

```
La factorizacion prima unica de 54 es: 2 * 3 * 3 * 3
```

Lecturas recomendadas

Abrahams, D. y A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond*. Boston: Addison-Wesley Profesional, 2004.

Ammeraal, L. *STL for C++ Programmers*. New York, NY: John Wiley & Sons, 1997.

Austern, M. H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Boston: Addison-Wesley, 2000.

Becker, P. *The C++ Standard Library Extensions: A Tutorial and Reference*. Boston: Addison-Wesley Professional, 2006.

Glass, G. y B. Shuchert. *The STL <Primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.

Heller, S. y Chrysalis Software Corp., *C++: A Dialog: Programming with the C++ Standard Library*. Nueva York, Prentice Hall PTR, 2002.

Josuttis, N. *The C++ Standard Library: A Tutorial and Reference (2nd edition)*. Boston: Addison-Wesley Professional, 2012.

Josuttis, N. *The C++ Standard Library: A Tutorial and Handbook*. Boston: Addison-Wesley, 2000.

Karlsson, B. *Beyond the C++ Standard Library: An Introduction to Boost.* Boston: Addison-Wesley Professional, 2005.

Koenig, A. y B. Moo. *Ruminations on C++*. Boston: Addison-Wesley, 1997.

Lippman, S., J. Lajoie y B. Moo. *C++ Primer (Fifth Edition)*. Boston: Addison-Wesley Professional, 2012.

Meyers, S. *Effective STL: 50 Specific Ways to Improve your Use of the Standard Template Library.* Boston: Addison-Wesley, 2001.

Musser, D.R., G. Derge y A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition*. Boston: Addison-Wesley, 2010.

Musser, D.R. y A. A. Stepanov. "Algorithm-Oriented Generic Libraries", *Software Practice and Experience,* vol. 24, núm. 7, julio de 1994.

Nelson, M. *C++ Programmer´s Guide to the Standard Template Library*. Foster City, CA: Programmer´s Press, 1995.

Pohl, I. *C++ Distilled: A Concise ANSI /ISO Reference and Style Guide*. Boston: Addison-Wesley, 1997.

Reese, G. *C++ Standard Library Practical Tips.* Hingham, MA: Charles River Media, 2005.

Robson, R. *Using the STL: The C++ Standard Template Library, Second Edition*. Nueva York: Springer, 2000.

Schildt, H. *STL Programming from the Ground Up, Third Edition*. Nueva York: McGraw-Hill Osborne Media, 2003.

Schildt, H. *STL Programming from the Ground Up*, Nueva York: Osborne McGraw-Hill, 1999.

Stepanov, A. y M. Lee. "The Standard Template Library", *Internet Distribution*. 31 de octubre de 1995 `<www.cs.rpi.edu/~musser/doc.ps>`.

Stroustrup, B. "C++11—the New ISO C++ Standard" `<www.stroustrup.com/C++11FAQ.html>`.

Stroustrup, B. "Making a `vector` Fit for a Standard", *The C++ Report*, octubre de 1994.

Stroustrup, B. *The Design and Evolution of C++*. Boston: Addison-Wesley, 1994.

Stroustrup, B. *The C++ Programming Language, Fourth Edition*. Boston: Addison-Wesley Professional, 2013.

Stroustrup, B. *The C++ Programming Language, Third Edition*. Boston: Addison-Wesley, 2000.

Vandevoorde, D. y N. Josuttis. *C++ Templates: The Complete Guide*. Boston: Addison-Wesley, 2003.

Vilot, M.J. "An Introduction to the Standard Template Library". *The C++ Report*, vol. 6, núm. 8, octubre de 1994.

Wilson, M. *Extended STL, Volume 1: Collections and Iterators*. Boston: Addison-Wesley, 2007.

16

Algoritmos de la Biblioteca estándar

El historiador es un profeta en reversa.

—Friedrich von Schlegel

Intenta hasta el final, y nunca te detengas ante la duda; Nada es tan difícil que no se pueda resolver mediante investigación.

—Robert Herrick

Objetivos

En este capítulo aprenderá a:

- Programar con muchas de las docenas de algoritmos de la Biblioteca estándar.
- Usar iteradores con algoritmos para acceder a (y manipular) los elementos de los contenedores de la Biblioteca estándar.
- Pasar apuntadores a funciones, objetos de funciones y expresiones lambda en algoritmos de la Biblioteca estándar.

Plan general

16.1 Introducción

En este capítulo seguimos nuestra discusión sobre los contenedores, iteradores y algoritmos de la Biblioteca estándar, enfocándonos en los algoritmos que realizan manipulaciones comunes de datos tales como *búsqueda*, *ordenamiento* y *comparación de elementos o contenedores enteros*. La Biblioteca estándar cuenta con más de 90 algoritmos, muchos de los cuales son nuevos en C++11. En las secciones 25 y 26.7 del documento del estándar de C++ encontrará la lista completa; además hay varias referencias en línea en donde podrá aprender más sobre cada algoritmo, como `en.cppreference.com/w/cpp/algorithm`. La mayoría de ellos utilizan iteradores para acceder a los elementos de un contenedor. Como veremos más adelante, varios algoritmos pueden recibir un *apuntador a función* (un apuntador al código de una función) como argumento. Dichos algoritmos utilizan el apuntador para llamar a la función; por lo general con uno o dos elementos de contenedores como argumentos. En este capítulo presentaremos los apuntadores a funciones con más detalle. Más adelante en el capítulo presentaremos el concepto de un *objeto de función*, que es similar a un apuntador a función, sólo que se implementa como objeto de una clase que tiene un operador de llamada a función sobrecargado (`operator()`), de modo que el objeto pueda usarse como el nombre de una función. Por último presentaremos las *expresiones lambda*: el nuevo mecanismo abreviado de C++11 para crear *objetos de función anónimos* (es decir, objetos de función que no tienen nombres).

16.2 Requisitos mínimos para los iteradores

Con pocas excepciones, la Biblioteca estándar separa algoritmos de contenedores. Esto facilita en gran medida el proceso de agregar nuevos algoritmos. Una parte importante de todo contenedor es el *tipo de iterador* que soporta (figura 15.7). Esto determina qué algoritmos pueden aplicarse al contenedor. Por ejemplo, los contenedores `vector` y `array` soportan *iteradores de acceso aleatorio* que proporcionan *todas* las operaciones de iteradores que se muestran en la figura 15.9. *Todos* los algoritmos de la Biblioteca estándar pueden operar con contenedores `vector` y los que no modifican el tamaño de un contenedor también pueden operar con contenedores `array`. Cada algoritmo de la Biblioteca estándar que recibe iteradores como argumentos requiere que esos iteradores proporcionen un mínimo nivel de funcionalidad. Por ejemplo, si un algoritmo requiere un *iterador de avance*, ese algoritmo puede operar sobre cualquier contenedor que soporte *iteradores de avance*, *iteradores bidireccionales* o *iteradores de acceso aleatorio*.

Observación de Ingeniería de Software 16.1

Los algoritmos de la Biblioteca estándar no *dependen de los detalles de implementación de los contenedores sobre los que operan. Mientras que los iteradores de un contenedor (o de un arreglo integrado) cumplan con los requisitos de un algoritmo, éste podrá trabajar en el contenedor.*

Tip de portabilidad 16.1

Puesto que los algoritmos de la Biblioteca estándar procesan contenedores sólo indirectamente *a través de iteradores, con frecuencia un algoritmo puede usarse con* muchos *contenedores diferentes.*

Observación de Ingeniería de Software 16.2

Los contenedores de la Biblioteca estándar se implementan de manera concisa. Los algoritmos se separan *de los contenedores y operan sobre elementos de éstos sólo* indirectamente *a través de iteradores. Esta separación facilita la escritura de algoritmos genéricos aplicables a una variedad de clases de contenedores.*

Observación de Ingeniería de Software 16.3

El uso del "iterador más débil" que produzca un rendimiento aceptable ayuda a producir componentes que pueden reutilizarse al máximo. Por ejemplo, si un algoritmo requiere sólo iteradores de avance, puede usarse con cualquier contenedor que soporte los iteradores de avance, iteradores bidireccionales o iteradores de acceso aleatorio. Sin embargo, un algoritmo que requiere iteradores de acceso aleatorio puede usarse sólo con contenedores que tengan iteradores de acceso aleatorio.

Invalidación de iteradores

Los iteradores simplemente *apuntan* a elementos de contenedores, por lo que es posible que los iteradores se vuelvan *inválidos* cuando ocurran ciertas modificaciones en los contenedores. Por ejemplo, si el programador invoca a `clear` en un `vector`, se *eliminan todos* sus elementos. Si un programa tuviera iteradores que apuntaran a los elementos de ese `vector` antes de llamar a `clear`, esos iteradores serían ahora *inválidos*. La sección 23 del estándar de C++ habla sobre todos los casos en los que los iteradores (y apuntadores y referencias) se invalidan para cada contenedor de la Biblioteca estándar. A continuación sintetizamos cuándo se invalidan los iteradores durante las operaciones de *inserción* y *eliminación*.

Al realizar operaciones de *inserción* en un:

- `vector`: si se reasigna el `vector`, se invalidan todos los iteradores que apuntan a ese `vector`. De lo contrario, se invalidan los iteradores desde el punto de inserción hasta el final del `vector`.
- `deque`: se invalidan todos los iteradores.
- `list` o `forward_list`: todos los iteradores *permanecen válidos*.
- Contenedor asociativo ordenado: todos los iteradores *permanecen válidos*.
- Contenedor asociativo desordenado: se invalidan todos los iteradores si hay que reasignar los contenedores.

Al *eliminar* datos de un contenedor, se invalidan los iteradores para los elementos *eliminados*. Además:

- `vector`: se invalidan los iteradores desde el elemento eliminado hasta el final del `vector`.
- `deque`: si se elimina un elemento en la parte media del `deque`, se invalidan todos los iteradores.

16.3 Algoritmos

Las secciones 16.3.1 a 16.3.13 demuestran muchos de los algoritmos de la Biblioteca estándar.

16.3.1 `fill`, `fill_n`, `generate` y `generate_n`

La figura 16.1 demuestra el uso de los algoritmos `fill`, `fill_n`, `generate` y `generate_n`. Los algoritmos **`fill`** y **`fill_n`** establecen todos los elementos en un rango de elementos del contenedor a un valor específico. Los algoritmos **`generate`** y **`generate_n`** usan una **función generadora** para crear valores para cada elemento en un *rango* de elementos del contenedor. La *función generadora* no toma argumentos y devuelve un valor que puede colocarse en un elemento del contenedor.

```
1  // Fig. 16.1: fig16_01.cpp
2  // Los algoritmos fill, fill_n, generate y generate_n.
3  #include <iostream>
4  #include <algorithm> // definiciones de los algoritmos
5  #include <array> // definición de la plantilla de clase array
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  char siguienteLetra(); // prototipo de la función generadora
10
11 int main()
12 {
13    array< char, 10 > chars;
14    ostream_iterator< char > salida( cout, " " );
15    fill( chars.begin(), chars.end(), '5' ); // llena chars con cincos
16
17    cout << "chars despues de llenarlo con cincos:\n";
18    copy( chars.cbegin(), chars.cend(), salida );
19
20    // llena los primeros cinco elementos de chars con As
21    fill_n( chars.begin(), 5, 'A' );
22
23    cout << "\n\nchars despues de llenar cinco elementos con As:\n";
24    copy( chars.cbegin(), chars.cend(), salida );
25
26    // genera los valores para todos los elementos de chars con siguienteLetra
27    generate( chars.begin(), chars.end(), siguienteLetra );
28
29    cout << "\n\nchars despues de generar las letras A-J:\n";
30    copy( chars.cbegin(), chars.cend(), salida );
31
32    // genera valores para los primeros cinco elementos de chars con siguienteLetra
33    generate_n( chars.begin(), 5, siguienteLetra );
34
35    cout << "\n\nchars despues de generar K-O para los"
36       << " primeros cinco elementos:\n";
37    copy( chars.cbegin(), chars.cend(), salida );
38    cout << endl;
39 } // fin de main
40
```

Fig. 16.1 | Los algoritmos `fill`, `fill_n`, `generate` y `generate_n` (parte 1 de 2).

```
41  // función generadora que devuelve la siguiente letra (empieza con A)
42  char siguienteLetra()
43  {
44     static char letra = 'A';
45     return letra++;
46  } // fin de la función siguienteLetra
```

```
chars despues de llenarlo con cincos:
5 5 5 5 5 5 5 5 5 5

chars despues de llenar cinco elementos con As:
A A A A A 5 5 5 5 5

chars despues de generar las letras A-J:
A B C D E F G H I J

chars despues de generar K-O para los primeros cinco elementos:
K L M N O F G H I J
```

Fig. 16.1 | Los algoritmos `fill`, `fill_n`, `generate` y `generate_n` (parte 2 de 2).

Algoritmo `fill`

En la línea 13 se define un `array` de 10 elementos que almacena valores `char`. En la línea 15 se utiliza el algoritmo `fill` para colocar el carácter `'5'` en cada elemento de `chars`, desde `chars.begin()` hasta, pero *sin* incluir a, `chars.end()`. Los iteradores suministrados como primer y segundo argumentos deben ser por lo menos *iteradores de avance* (es decir, que puedan utilizarse tanto como entrada desde un contenedor como de salida hacia un contenedor, en dirección *hacia delante*).

Algoritmo `fill_n`

En la línea 21 se utiliza el algoritmo `fill_n` para colocar el carácter `'A'` en los primeros cinco elementos de `chars`. El iterador suministrado como primer argumento debe ser por lo menos un *iterador de salida* (es decir, que pueda utilizarse para *escribir* en un contenedor en dirección *hacia delante*). El segundo argumento especifica el número de elementos a llenar. El tercer argumento especifica el valor a colocar en cada elemento.

Algoritmo `generate`

En la línea 27 se utiliza el algoritmo `generate` para colocar el resultado de una llamada a la *función generadora* `siguienteLetra` en cada elemento de `chars`, desde `chars.begin()` hasta, pero *sin* incluir a, `chars.end()`. Los iteradores suministrados como primer y segundo argumentos deben ser por lo menos *iteradores de avance*. La función `siguienteLetra` (líneas 42 a 46) empieza con el carácter `'A'` que se mantiene en una variable local `static`. La instrucción en la línea 45 post-incrementa el valor de `letra` y devuelve su valor anterior cada vez que se hace una llamada a `siguienteLetra`.

Algoritmo `generate_n`

En la línea 33 se utiliza el algoritmo `generate_n` para colocar el resultado de una llamada a la *función generadora* `siguienteLetra` en cinco elementos de `chars`, empezando desde `chars.begin()`. El iterador que se suministra como el primer argumento debe ser por lo menos un *iterador de salida*.

Una nota con respecto a la lectura de la documentación de los algoritmos de la Biblioteca estándar

Cuando lea en la documentación de los algoritmos de la Biblioteca estándar sobre los algoritmos que pueden recibir apuntadores a funciones como argumentos, observará que los parámetros correspon-

dientes *no* muestran declaraciones de apuntadores. Dichos parámetros en realidad pueden recibir como parámetros *apuntadores a funciones*, *objetos de funciones* (sección 16.4) o *expresiones lambda* (sección 16.5). Por esta razón, la Biblioteca estándar declara dichos parámetros utilizando nombres más genéricos.

Por ejemplo, el prototipo del algoritmo `generate` se lista en el documento del estándar de C++ como:

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
  Generator gen);
```

para indicar que `generate` espera objetos *`ForwardIterator`* como argumentos, los cuales representan el rango de elementos a procesar, además de una *función `Generator`*. El estándar explica que el algoritmo llama a la función `Generator` para obtener un valor para cada elemento en el rango especificado por los objetos *`ForwardIterator`*. El estándar también especifica que la función `Generator` no debe recibir argumentos y debe devolver un valor del tipo del elemento.

Hay documentación similar disponible para cada algoritmo que pueda recibir un apuntador a función, objeto de función o expresión lambda. En la mayoría de los ejemplos de este capítulo, al momento de presentar cada algoritmo especificamos los requisitos para dichos parámetros. Por lo general lo hacemos en el contexto de las funciones y pasamos apuntadores a funciones a los algoritmos. En las secciones 16.4 y 16.5 veremos cómo crear y usar objetos de funciones y expresiones lambda que se puedan pasar a los algoritmos.

16.3.2 `equal`, `mismatch` y `lexicographical_compare`

En la figura 16.2 se demuestra la comparación de secuencias de valores para ver si son iguales mediante el uso de los algoritmos `equal`, `mismatch` y `lexicographical_compare`.

```
1  // Fig. 16.2: fig16_02.cpp
2  // Los algoritmos equal, mismatch y lexicographical_compare.
3  #include <iostream>
4  #include <algorithm> // definiciones de los algoritmos
5  #include <array> // definición de la plantilla de clase array
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  int main()
10 {
11    const size_t TAMANIO = 10;
12    array< int, TAMANIO > a1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13    array< int, TAMANIO > a2( a1 ); // inicializa a2 com uma copia de a1
14    array< int, TAMANIO > a3 = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
15    ostream_iterator< int > salida( cout, " " );
16
17    cout << "a1 contiene: ";
18    copy( a1.cbegin(), a1.cend(), salida );
19    cout << "\na2 contiene: ";
20    copy( a2.cbegin(), a2.cend(), salida );
21    cout << "\na3 contiene: ";
22    copy( a3.cbegin(), a3.cend(), salida );
```

Fig. 16.2 | Los algoritmos `equal`, `mismatch` y `lexicographical_compare` (parte 1 de 2).

```
23
24    // compara a1 y a2 para ver si son iguales
25    bool resultado = equal( a1.cbegin(), a1.cend(), a2.cbegin() );
26    cout << "\n\na1 " << ( resultado ? "es" : "no es" )
27       << " igual a a2.\n";
28
29    // compara los vectores a1 y a3 para ver si son iguales
30    resultado = equal( a1.cbegin(), a1.cend(), a3.cbegin() );
31    cout << "a1 " << ( resultado ? "es" : "no es" ) << " igual a a3.\n";
32
33    // comprueba que no haya inconsistencia entre a1 y a3
34    auto ubicacion = mismatch( a1.cbegin(), a1.cend(), a3.cbegin() );
35    cout << "\nHay una inconsistencia entre a1 y a3 en la ubicacion "
36       << ( ubicacion.first - a1.begin() ) << "\nen donde a1 contiene "
37       << *ubicacion.first << " y a3 contiene "  << *ubicacion.second
38       << "\n\n";
39
40    char c1[ TAMANIO ] = "HOLA";
41    char c2[ TAMANIO ] = "BYE BYE";
42
43    // realiza una comparación lexicográfica de c1 y c2
44    resultado = lexicographical_compare(
45       begin( c1 ), end( c1 ), begin( c2 ), end( c2 ) );
46    cout << c1 << ( resultado ? " es menor que " :
47       " es mayor o igual a " )  << c2 << endl;
48 } // fin de main
```

```
a1 contiene: 1 2 3 4 5 6 7 8 9 10
a2 contiene: 1 2 3 4 5 6 7 8 9 10
a3 contiene: 1 2 3 4 1000 6 7 8 9 10

a1 es igual a a2.
a1 no es igual a a3.

Hay una inconsistencia entre a1 y a3 en la ubicacion 4
en donde a1 contiene 5 y a3 contiene 1000

HOLA es mayor o igual a BYE BYE
```

Fig. 16.2 | Los algoritmos `equal`, `mismatch` y `lexicographical_compare` (parte 2 de 2).

Algoritmo `equal`

La línea 25 utiliza el algoritmo **`equal`** para comparar dos secuencias de valores y ver si son iguales. La segunda secuencia debe contener por lo menos tantos elementos como la primera: `equal` devuelve `false` si las secuencias *no* son de la misma longitud. La función `operator==` (ya sea integrada o sobrecargada) realiza las comparaciones de los elementos. En este ejemplo se comparan los elementos en `a1` desde `a1.cbegin()` hasta (pero *sin* incluir a) `a1.cend()` con los elementos en `a2` empezando desde `a2.cbegin()`. En este ejemplo, `a1` y `a2` son iguales. Los tres iteradores que se toman como argumentos deben ser por lo menos *iteradores de entrada* (es decir, que puedan utilizarse para introducir valores desde una secuencia en dirección *hacia delante*). En la línea 30 se utiliza a la función `equal` para comparar `a1` y `a3`, que *no* son iguales.

Algoritmo `equal` *con función predicado binaria*

Hay otra versión de la función `equal` que toma una *función predicado binaria* como cuarto parámetro. Esta función recibe los dos elementos que se van a comparar y devuelve un valor `bool` que indica si los

elementos son iguales o no. Esto puede ser útil en secuencias que almacenan objetos o apuntadores a valores, en vez de los valores actuales, ya que se pueden definir una o más comparaciones. Por ejemplo, puede comparar objetos `Empleado` en base a la edad, número de seguro social o ubicación en vez de comparar objetos completos. Puede comparar a qué hacen referencia los apuntadores, en vez de comparar sus valores (es decir, las direcciones almacenadas en los apuntadores).

Algoritmo ***`mismatch`***

En la línea 34 se hace una llamada a la función **`mismatch`** para comparar dos secuencias de valores. El algoritmo devuelve un objeto `pair` de iteradores que indican la posición en cada secuencia de los elementos *inconsistentes*. Si todos los elementos concuerdan, los dos iteradores en el objeto `pair` son iguales al último iterador para cada secuencia. Los tres iteradores que se toman como argumentos deben ser por lo menos *iteradores de entrada*. *Inferimos* el tipo del objeto `pair` llamado `ubicacion` con la palabra clave `auto` de C++11 (línea 34). En la línea 36 se determina la ubicación actual de la inconsistencia en los objetos `array` mediante la expresión `ubicacion.first - a1.begin()`, cuyo resultado es el número de elementos entre los iteradores (esto es análogo a la aritmética de apuntadores; capítulo 8). Esto corresponde al número de elementos en este ejemplo, ya que la comparación se realiza desde el inicio de cada `array`. Al igual que con `equal`, hay otra versión de `mismatch` que recibe una *función predicado binaria* como cuarto parámetro.

Algoritmo ***`lexicographical_compare`***

En las líneas 44 y 45 se utiliza el algoritmo **`lexicographical_compare`** para comparar el contenido de dos arreglos `char` integrados. Los cuatro iteradores que se toman como argumentos en este algoritmo deben ser por lo menos *iteradores de entrada*. Como usted sabe, los apuntadores en arreglos integrados son *iteradores de acceso aleatorio*. Los primeros dos argumentos de iteradores especifican el rango de ubicaciones en la primera secuencia. Los últimos dos especifican el rango de las ubicaciones en la segunda secuencia. Una vez más, utilizamos las funciones `begin` y `end` de C++11 para determinar el rango de elementos para cada arreglo integrado. Al iterar a través de las secuencias, `lexicographical_compare` comprueba si el elemento en la primera secuencia es menor que el elemento correspondiente en la segunda secuencia. De ser así, el algoritmo devuelve `true`. Si el elemento en la primera secuencia es mayor o igual que el elemento en la segunda, el algoritmo devuelve `false`. Este algoritmo se puede utilizar para ordenar las secuencias en forma *lexicográfica*. Por lo general, dichas secuencias contienen cadenas.

16.3.3 `remove`, `remove_if`, `remove_copy` y `remove_copy_if`

La figura 16.3 demuestra cómo eliminar valores de una secuencia mediante los algoritmos `remove`, `remove_if`, `remove_copy` y `remove_copy_if`.

```
1  // Fig. 16.3: fig16_03.cpp
2  // Los algoritmos remove, remove_if, remove_copy y remove_copy_if.
3  #include <iostream>
4  #include <algorithm> // definiciones de los algoritmos
5  #include <array> // definición de la plantilla de clase array
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  bool mayor9( int ); // prototipo
10
```

Fig. 16.3 | Los algoritmos `remove`, `remove_if`, `remove_copy` y `remove_copy_if` (parte 1 de 3).

```
11 int main()
12 {
13    const size_t TAMANIO = 10;
14    array< int, TAMANIO > inic = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
15    ostream_iterator< int > salida( cout, " " );
16
17    array< int, TAMANIO > a1( inic ); // inicializa con una copia de inic
18    cout << "a1 antes de eliminar todos los numeros 10:\n   ";
19    copy( a1.cbegin(), a1.cend(), salida );
20
21    // elimina todos los numeros 10 de a1
22    auto nuevoUltimoElemento = remove( a1.begin(), a1.end(), 10 );
23    cout << "\na1 despues de eliminar todos los numeros 10:\n   ";
24    copy( a1.begin(), nuevoUltimoElemento, salida );
25
26    array< int, TAMANIO > a2( inic ); // inicializa con copia de inic
27    array< int, TAMANIO > c = { 0 }; // inicializa con ceros
28    cout << "\n\na2 antes de eliminar todos los numeros 10 y copiar:\n   ";
29    copy( a2.cbegin(), a2.cend(), salida );
30
31    // copia de a2 a c, eliminando los numeros 10 en el proceso
32    remove_copy( a2.cbegin(), a2.cend(), c.begin(), 10 );
33    cout << "\nc despues de eliminar todos los numeros 10 de a2:\n   ";
34    copy( c.cbegin(), c.cend(), salida );
35
36    array< int, TAMANIO > a3( inic ); // inicializa con copia de inic
37    cout << "\n\na3 antes de eliminar todos los elementos mayores que 9:\n   ";
38    copy( a3.cbegin(), a3.cend(), salida );
39
40    // elimina los elementos mayores que 9 de a3
41    nuevoUltimoElemento = remove_if( a3.begin(), a3.end(), mayor9 );
42    cout << "\na3 despues de eliminar todos los elementos mayores que 9:\n   ";
43    copy( a3.begin(), nuevoUltimoElemento, salida );
44
45    array< int, TAMANIO > a4( inic ); // inicializa con copia de inic
46    array< int, TAMANIO > c2 = { 0 }; // inicializa con ceros
47    cout << "\n\na4 antes de eliminar todos los elementos"
48       << "\nmayores que 9 y copiar:\n   ";
49    copy( a4.cbegin(), a4.cend(), salida );
50
51    // copia elementos de a4 a c2, eliminando los elementos
52    // mayores que 9 en el proceso
53    remove_copy_if( a4.cbegin(), a4.cend(), c2.begin(), mayor9 );
54    cout << "\nc2 despues de eliminar todos los elementos"
55       << "\nmayores que 9 de a4:\n   ";
56    copy( c2.cbegin(), c2.cend(), salida );
57    cout << endl;
58 } // fin de main
59
60 // determina si el argumento es mayor que 9
61 bool mayor9( int x )
62 {
```

Fig. 16.3 | Los algoritmos `remove`, `remove_if`, `remove_copy` y `remove_copy_if` (parte 2 de 3).

```
63     return x > 9;
64   } // fin de la función mayor9
```

```
a1 antes de eliminar todos los numeros 10:
   10 2 10 4 16 6 14 8 12 10
a1 despues de eliminar todos los numeros 10:
   2 4 16 6 14 8 12

a2 antes de eliminar todos los numeros 10 y copiar:
   10 2 10 4 16 6 14 8 12 10
c despues de eliminar todos los numeros 10 de a2:
   2 4 16 6 14 8 12 0 0 0

a3 antes de eliminar todos los elementos mayores que 9:
   10 2 10 4 16 6 14 8 12 10
a3 despues de eliminar todos los elementos mayores que 9:
   2 4 6 8

a4 antes de eliminar todos los elementos
mayores que 9 y copiar:
   10 2 10 4 16 6 14 8 12 10
c2 despues de eliminar todos los elementos
mayores que 9 de a4:
   2 4 6 8 0 0 0 0 0 0
```

Fig. 16.3 | Los algoritmos `remove`, `remove_if`, `remove_copy` y `remove_copy_if` (parte 3 de 3).

Algoritmo `remove`

En la línea 22 se utiliza el algoritmo **`remove`** para eliminar *todos* los elementos con el valor `10` en el rango desde `a1.begin()` hasta, pero *sin* incluir a, `a1.end()` de `a1`. Los primeros dos iteradores que se reciben como argumentos deben ser *iteradores de avance*. Este algoritmo *no* modifica el número de elementos en el contenedor ni destruye los elementos eliminados, pero sí desplaza a *todos* los elementos que *no* se eliminen hacia el *inicio* del contenedor. El algoritmo devuelve un iterador colocado después del último elemento que no se haya eliminado. Los elementos que se encuentran desde la posición del iterador hasta el final del contenedor tienen *valores indefinidos*.

Algoritmo `remove_copy`

En la línea 32 se utiliza el algoritmo **`remove_copy`** para copiar *todos* los elementos que *no* tengan el valor `10` que se encuentren en el rango desde `a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()` de `a2`. Los elementos se colocan en `c`, empezando en la posición `c.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser *iteradores de entrada*. El iterador que se suministra como tercer argumento debe ser un *iterador de salida*, para que el elemento que se va a copiar pueda *insertarse* en la ubicación de copia. Este algoritmo devuelve un iterador que está colocado después del último elemento copiado al `array c`.

Algoritmo `remove_if`

En la línea 41 se utiliza el algoritmo **`remove_if`** para eliminar *todos* aquellos elementos en el rango de `a3.begin()` hasta, pero *sin* incluir a, `a3.end()` de `a3`, para lo cual nuestra *función predicado* unaria `mayor9` definida por el usuario devuelve `true`. Esta función (definida en las líneas 61 a 64) devuelve `true` si el valor que recibe es mayor que 9; en caso contrario devuelve `false`. Los iteradores suministrados como los primeros dos argumentos deben ser *iteradores de avance*. Este algoritmo *no* modifica el número de elementos en el contenedor, pero sí desplaza al *inicio* del contenedor todos los elementos que *no*

se eliminan. Este algoritmo devuelve un iterador que se coloca después del último elemento que *no* se haya eliminado. Todos los elementos a partir de la posición del iterador y hasta el final del contenedor tienen valores *indefinidos*.

Algoritmo **`remove_copy_if`**

En la línea 53 se utiliza el algoritmo **`remove_copy_if`** para copiar todos aquellos elementos en el rango desde `a4.cbegin()` hasta, pero *sin* incluir a, `a4.cend()` de `a4` para los que la *función predicado unaria* `mayor9` devuelva `true`. Los elementos se colocan en `c2`, empezando en la posición `c2.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser *iteradores de entrada*. El iterador que se suministra como tercer argumento debe ser un *iterador de salida*, para que el elemento que se va a copiar pueda *asignarse* a la ubicación de copia. Este algoritmo devuelve un iterador que se coloca después del último elemento copiado en `c2`.

16.3.4 `replace`, `replace_if`, `replace_copy` y `replace_copy_if`

La figura 16.4 demuestra cómo reemplazar valores de una secuencia mediante los algoritmos `replace`, `replace_if`, `replace_copy` y `replace_copy_if`.

```
1  // Fig. 16.4: fig16_04.cpp
2  // Los algoritmos replace, replace_if, replace_copy y replace_copy_if.
3  #include <iostream>
4  #include <algorithm>
5  #include <array>
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  bool mayor9( int ); // prototipo de la función predicado
10
11 int main()
12 {
13    const size_t TAMANIO = 10;
14    array< int, TAMANIO > inic =  = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
15    ostream_iterator< int > salida( cout, " " );
16
17    array< int, TAMANIO >  a1( inic ); // inicializa con copia de inic
18    cout << "a1 antes de reemplazar todos los numeros 10:\n   ";
19    copy( a1.cbegin(), a1.cend(), salida );
20
21    // reemplaza todos los numeros 10 en a1 con 100
22    replace( a1.begin(), a1.end(), 10, 100 );
23    cout << "\na1 despues de reemplazar los numeros 10 con 100:\n   ";
24    copy( a1.cbegin(), a1.cend(), salida );
25
26    array< int, TAMANIO > a2( inic ); // inicializa con copia de inic
27    array< int, TAMANIO > c1; // crea una instancia de c1
28    cout << "\n\na2 antes de reemplazar todos los numeros 10 y copiar:\n   ";
29    copy( a2.cbegin(), a2.cend(), salida );
30
31    // copia de a2 a c1, reemplazando los numeros 10 con 100
32    replace_copy( a2.cbegin(), a2.cend(), c1.begin(), 10, 100 );
```

Fig. 16.4 | Algoritmos `replace`, `replace_if`, `replace_copy` y `replace_copy_if` (parte 1 de 2).

```
33     cout << "\nc1 despues de reemplazar todos los numeros 10 en a2:\n   ";
34     copy( c1.cbegin(), c1.cend(), salida );
35
36     array< int, TAMANIO > a3( inic ); // inicializa con copia de inic
37     cout << "\n\na3 antes de reemplazar valores mayores que 9:\n   ";
38     copy( a3.cbegin(), a3.cend(), salida );
39
40     // reemplaza los valores mayores que 9 en a3 con 100
41     replace_if( a3.begin(), a3.end(), mayor9, 100 );
42     cout << "\na3 despues de reemplazar todos los valores"
43        << "\nmayores que 9 con 100:\n   ";
44     copy( a3.cbegin(), a3.cend(), salida );
45
46     array< int, TAMANIO > a4( inic ); // inicializa con copia de inic
47     array< int, TAMANIO > c2; // crea instancia de c2
48     cout << "\n\na4 antes de reemplazar todos los valores mayores "
49        << "que 9 y copiar:\n   ";
50     copy( a4.cbegin(), a4.cend(), salida );
51
52     // copia a4 a c2, reemplazando los elementos mayores que 9 con 100
53     replace_copy_if( a4.cbegin(), a4.cend(), c2.begin(), mayor9, 100 );
54     cout << "\nc2 despues de reemplazar todos los valores mayores que 9 en v4:\n   ";
55     copy( c2.begin(), c2.end(), salida );
56     cout << endl;
57  } // fin de main
58
59  // determina si el argumento es mayor que 9
60  bool mayor9( int x )
61  {
62     return x > 9;
63  } // fin de la función mayor9
```

```
a1 antes de reemplazar todos los numeros 10:
   10 2 10 4 16 6 14 8 12 10
a1 despues de reemplazar los numeros 10 con 100:
   100 2 100 4 16 6 14 8 12 100

a2 antes de reemplazar todos los numeros 10 y copiar:
   10 2 10 4 16 6 14 8 12 10
c1 despues de reemplazar todos los numeros 10 en a2:
   100 2 100 4 16 6 14 8 12 100

a3 antes de reemplazar valores mayores que 9:
   10 2 10 4 16 6 14 8 12 10
a3 despues de reemplazar todos los valores
mayores que 9 con 100:
   100 2 100 4 100 6 100 8 100 100

a4 antes de reemplazar todos los valores mayores que 9 y copiar:
   10 2 10 4 16 6 14 8 12 10
c2 despues de reemplazar todos los valores mayores que 9 en a4:
   100 2 100 4 100 6 100 8 100 100
```

Fig. 16.4 | Algoritmos `replace`, `replace_if`, `replace_copy` y `replace_copy_if` (parte 2 de 2).

Algoritmo **`replace`**
La línea 22 utiliza el algoritmo **`replace`** para reemplazar *todos* los elementos con el valor `10` en el rango de `a1.begin()` hasta, pero *sin* incluir a, `a1.end()` con el nuevo valor `100`. Los iteradores que se suministran como los primeros dos argumentos deben ser *iteradores de avance*, de manera que el algoritmo pueda *modificar* los elementos en la secuencia.

Algoritmo **`replace_copy`**
En la línea 32 se utiliza el algoritmo **`replace_copy`** para copiar todos los elementos en el rango de `a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()`, reemplazando a *todos* los elementos que tienen el valor `10` con el nuevo valor `100`. Los elementos se copian en `c1`, empezando en la posición `c1.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser *iteradores de entrada*. El iterador que se suministra como el tercer argumento debe ser un *iterador de salida*, para que el elemento que va a copiarse pueda *asignarse* a la ubicación de copia. Esta función devuelve un iterador que se coloca después del último elemento copiado en `c1`.

Algoritmo **`replace_if`**
En la línea 41 se utiliza el algoritmo **`replace_if`** para reemplazar a *todos* aquellos elementos en el rango de `a3.begin()` hasta, pero *sin* incluir a, `a3.end()` para el que la *función predicado unaria* `mayor9` devuelva `true`. Esta función (definida en las líneas 60 a 63) devuelve `true` si el valor que recibe es mayor que 9; en cualquier otro caso devuelve `false`. El valor `100` sustituye a cualquier valor mayor de `9`. Los iteradores que se suministran como los primeros dos argumentos deben ser *iteradores de avance*.

Algoritmo **`replace_copy_if`**
En la línea 53 se utiliza el algoritmo **`replace_copy_if`** para copiar *todos* los elementos en el rango que empieza desde `a4.cbegin()` hasta, pero *sin* incluir a `a4.cend()`. Los elementos para los que la *función predicado unaria* `mayor9` devuelva `true` se reemplazan con el valor `100`. Los elementos se colocan en `c2`, empezando en la posición `c2.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser *iteradores de entrada*. El iterador que se suministra como el tercer argumento debe ser un *iterador de salida*, para que el elemento que va a copiarse pueda *asignarse* a la ubicación de copia. Este algoritmo devuelve un iterador que se coloca después del último elemento copiado en `c2`.

16.3.5 Algoritmos matemáticos

La figura 16.5 demuestra varios algoritmos matemáticos comunes: `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `minmax_element`, `acumulate`, `for_each` y `transform`.

```
1  // Fig. 16.5: fig16_05.cpp
2  // Algoritmos matemáticos de la Biblioteca estándar.
3  #include <iostream>
4  #include <algorithm> // definiciones de algoritmos
5  #include <numeric> // aquí se define accumulate
6  #include <array>
7  #include <iterator>
8  using namespace std;
9
10 bool mayor9( int ); // prototipo de la función predicado
```

Fig. 16.5 | Algoritmos matemáticos de la Biblioteca estándar (parte 1 de 3).

```
11 void imprimirCuadrado( int ); // imprime el cuadrado de un valor
12 int calcularCubo( int ); // calcula el cubo de un valor
13
14 int main()
15 {
16    const size_t TAMANIO = 10;
17    array< int, TAMANIO > a1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
18    ostream_iterator< int > salida( cout, " " );
19
20    cout << "a1 antes de random_shuffle: ";
21    copy( a1.cbegin(), a1.cend(), salida );
22
23    random_shuffle( a1.begin(), a1.end() ); // revuelve los elementos de a1
24    cout << "\na1 después de random_shuffle: ";
25    copy( a1.cbegin(), a1.cend(), salida );
26
27    array< int, TAMANIO > a2 = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
28    cout << "\n\na2 contiene: ";
29    copy( a2.cbegin(), a2.cend(), salida );
30
31    // cuenta el número de elementos en a2 con el valor 8
32    int resultado = count( a2.cbegin(), a2.cend(), 8 );
33    cout << "\nNumero de elementos que concuerdan con 8: " << resultado;
34
35    // cuenta el número de elementos en a2 mayores que 9
36    resultado = count_if( a2.cbegin(), a2.cend(), mayor9 );
37    cout << "\nNumero de elementos mayores que 9: " << resultado;
38
39    // localiza el elemento mínimo en a2
40    cout << "\n\nEl elemento minimo en el vector a2 es: "
41       << *( min_element( a2.cbegin(), a2.cend() ) );
42
43    // localiza el elemento máximo en a2
44    cout << "\nEl elemento maximo en a2 es: "
45       << *( max_element( a2.cbegin(), a2.cend() ) );
46
47    // localiza los elementos mínimo y máximo en a2
48    auto minYMax = minmax_element( a2.cbegin(), a2.cend() );
49    cout << "\nLos elementos minimo y maximo en a2 son "
50       << *minYMax.first << " y " << *minYMax.second
51       << ", respectivamente";
52
53    // calcula la suma de los elementos en a1
54    cout << "\n\nEl total de los elementos en a1 es: "
55       << accumulate( a1.cbegin(), a1.cend(), 0 ) ;
56
57    // imprime el cuadrado de cada elemento en a1
58    cout << "\n\nEl cuadrado de cada entero en a1 es:\n";
59    for_each( a1.cbegin(), a1.cend(), imprimirCuadrado );
60
61    array< int, TAMANIO > cubos; // crea instancia de cubos
62
```

Fig. 16.5 | Algoritmos matemáticos de la Biblioteca estándar (parte 2 de 3).

```
63     // calcula el cubo de cada elemento en a1; coloca los resultados en cubos
64     transform( a1.cbegin(), a1.cend(), cubos.begin(), calcularCubo );
65     cout << "\n\nEl cubo de cada entero en a1 es:\n";
66     copy( cubos.cbegin(), cubos.cend(), salida );
67     cout << endl;
68  } // fin de main
69
70  // determina si el argumento es mayor que 9
71  bool mayor9( int valor )
72  {
73     return valor > 9;
74  } // fin de la función mayor9
75
76  // imprime el cuadrado del argumento
77  void imprimirCuadrado( int valor )
78  {
79     cout << valor * valor << ' ';
80  } // fin de la función imprimirCuadrado
81
82  // devuelve el cubo del argumento
83  int calcularCubo( int valor )
84  {
85     return valor * valor * valor;
86  } // fin de la función calcularCubo
```

```
a1 antes de random_shuffle: 1 2 3 4 5 6 7 8 9 10
a1 despues de random_shuffle: 9 2 10 3 1 6 8 4 5 7

a2 contiene: 100 2 8 1 50 3 8 8 9 10
Numero de elementos que concuerdan con 8: 3
Numero de elementos mayores que 9: 3

El elemento minimo en a2 es: 1
El elemento maximo en a2 es: 100
Los elementos minimo y maximo en a2 son 1 y 100, respectivamente

El total de los elementos en a1 es: 55

El cuadrado de cada entero en a1 es:
81 4 100 9 1 36 64 16 25 49

El cubo de cada entero en a1 es:
729 8 1000 27 1 216 512 64 125 343
```

Fig. 16.5 | Algoritmos matemáticos de la Biblioteca estándar (parte 3 de 3).

Algoritmo `random_shuffle`

En la línea 23 se utiliza el algoritmo `random_shuffle` para reordenar en forma aleatoria los elementos en el rango empezando desde `a1.begin()` hasta, pero sin incluir a, `a1.end()`. Este algoritmo toma *dos iteradores de acceso aleatorio* como argumentos. Esta versión de `random_shuffle` utiliza `rand` para la randomización y produce los mismos resultados cada vez que se ejecuta el programa, a menos que sembremos el generador de números aleatorios con `srand`. Otra versión de `random_shuffle` recibe como su tercer argumento un generador de números aleatorios uniforme de C++11.

Algoritmo `count`

En la línea 32 se utiliza el algoritmo `count` para contar los elementos que tengan el valor de `8` en el rango que empieza desde `a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()`. Este algoritmo requiere que sus dos argumentos iteradores sean por lo menos *iteradores de entrada*.

Algoritmo `count_if`

En la línea 36 se utiliza el algoritmo `count_if` para contar los elementos en el rango desde `a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()` para los que la *función predicado* `mayor9` devuelva `true`. El algoritmo `count_if` requiere que sus dos argumentos iteradores sean por lo menos *iteradores de entrada.*

Algoritmo `min_element`

En la línea 41 se utiliza el algoritmo `min_element` para localizar el *menor* elemento en el rango que empieza desde `a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()`. El algoritmo devuelve un *iterador de avance* localizado en el *primer* elemento que sea menor, o `a2.end()` si el rango está *vacío*. El algoritmo requiere que sus dos argumentos iteradores sean por lo menos *iteradores de avance*. Una segunda versión de este algoritmo toma como tercer argumento una función binaria que compara dos elementos en la secuencia. Este algoritmo devuelve el valor `bool true` si el primer argumento es *menor que* el segundo.

Tip para prevenir errores 16.1

Es una buena práctica comprobar que el rango especificado en una llamada a `min_element` *no esté vacío y comprobar también que el valor de retorno no sea el iterador que está "más allá del final".*

Algoritmo `max_element`

En la línea 45 se utiliza el algoritmo `max_element` para localizar el *mayor* elemento en el rango que empieza desde `a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()`. El algoritmo devuelve un *iterador de avance* que se coloca en el *primer* elemento más grande. Este algoritmo requiere que sus dos argumentos iteradores sean por lo menos *iteradores de avance*. Una segunda versión de este algoritmo toma como su tercer argumento a una *función predicado binaria* que compara los elementos en la secuencia. La función binaria toma dos argumentos y devuelve el valor `bool true` si el primer argumento es *menor que* el segundo.

C++11: algoritmo `minmax_element`

En la línea 48 se utiliza el nuevo algoritmo `minmax_element` de C++11 para localizar los elementos *menor* y *mayor* en el rango que empieza desde `a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()`. El algoritmo devuelve un par de *iteradores de avance* ubicados en los elementos menor y mayor, respectivamente. Si hay elementos menor o mayor duplicados, los iteradores se ubican en el primer valor menor y en el último valor mayor. Los dos argumentos iteradores del algoritmo deben ser al menos *iteradores de avance*. Una segunda versión de este algoritmo recibe como su tercer argumento una *función predicado binaria* que compara los elementos en la secuencia. La función binaria toma dos argumentos y devuelve el valor `bool true` si el primer argumento es *menor que* el segundo.

Algoritmo `accumulate`

En la línea 55 se utiliza el algoritmo `accumulate` (cuya plantilla se encuentra en el encabezado `<numeric>`) para sumar los valores en el rango que empieza desde `a1.cbegin()` hasta, pero *sin* incluir a, `a1.cend()`. Los dos argumentos iteradores del algoritmo deben ser por lo menos *iteradores de entrada* y su tercer argumento representa el valor inicial del total. Una segunda versión de este algoritmo toma como cuarto argumento una función general que determina cómo se acumulan los elementos. Esta función general debe tomar *dos* argumentos y devolver un resultado. El primer argumento de esta función

es el valor actual de la acumulación. El segundo argumento es el valor del elemento actual en la secuencia que se va a acumular.

Algoritmo `for_each`

En la línea 59 se utiliza el algoritmo **`for_each`** para aplicar una función general a cada elemento en el rango que empieza desde `a1.cbegin()` hasta, pero sin incluir a, `a1.cend()`. La función general toma el elemento actual como argumento y puede modificarlo (si se recibe por referencia y no es `const`). El algoritmo `for_each` requiere que sus dos argumentos iteradores sean por lo menos *iteradores de entrada*.

Algoritmo `transform`

En la línea 63 se utiliza el algoritmo **`transform`** para aplicar una función general a *cada* elemento en el rango que empieza desde `a1.cbegin()` hasta, pero *sin* incluir a, `a1.cend()`. La función general (el cuarto argumento) debe tomar el elemento actual como argumento, *no* debe modificarlo y debe regresar el valor transformado (mediante `transform`). El algoritmo `transform` requiere que sus primeros dos argumentos iteradores sean por lo menos *iteradores de entrada* y que su tercer argumento sea por lo menos un *iterador de salida*. El tercer argumento especifica en dónde deben colocarse los valores transformados. Observe que el tercer argumento puede ser igual al primero. Otra versión de `transform` acepta cinco argumentos: los primeros dos argumentos son *iteradores de entrada* que especifican un rango de elementos de un contenedor de origen, el tercer argumento es un *iterador de entrada* que especifica el primer elemento en otro contenedor de origen, el cuarto argumento es un *iterador de salida* que especifica en dónde se deben colocar los valores transformados, y el último argumento es una función general que recibe dos argumentos. Esta versión de `transform` toma un elemento de cada uno de los dos orígenes y aplica la función general a ese par de elementos, y después coloca el valor transformado en la ubicación especificada por el cuarto argumento.

16.3.6 Algoritmos básicos de búsqueda y ordenamiento

La figura 16.6 demuestra algunos de los algoritmos básicos de búsqueda y ordenamiento de la Biblioteca estándar: `find`, `find_if`, `sort`, `binary_search`, `all_of`, `any_of`, `none_of` y `find_if_not`.

```
1  // Fig. 16.6: fig16_06.cpp
2  // Algoritmos de búsqueda y ordenamiento de la Biblioteca estándar.
3  #include <iostream>
4  #include <algorithm> // definiciones de los algoritmos
5  #include <array> // definición de la plantilla de clase array
6  #include <iterator>
7  using namespace std;
8
9  bool mayor10( int valor ); // prototipo de la función predicado
10
11 int main()
12 {
13    const size_t TAMANIO = 10;
14    array< int, TAMANIO > a = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
15    ostream_iterator< int > salida( cout, " " );
```

Fig. 16.6 | Algoritmos básicos de búsqueda y ordenamiento de la Biblioteca estándar (parte 1 de 3).

```
16
17    cout << "El arreglo a contiene: ";
18    copy( a.cbegin(), a.cend(), salida ); // muestra el vector de salida
19
20    // localiza la primera ocurrencia de 16 en a
21    auto ubicacion = find( a.cbegin(), a.cend(), 16 );
22
23    if ( ubicacion != a.cend() ) // encontró el 16
24       cout << "\n\nSe encontro el 16 en la ubicacion "  << ( ubicacion -
                                                               a.cbegin() );
25    else // no se encontró el 16
26       cout << "\n\nNo se encontro el 16";
27
28    // localiza la primera ocurrencia de 100 en a
29    ubicacion = find( a.cbegin(), a.cend(), 100 );
30
31    if ( ubicacion != a.cend() ) // encontró el 100
32      cout << "\nSe encontro el 100 en la ubicacion "  << ( ubicacion - a.cbegin() );
33    else // no se encontró el 100
34       cout << "\nNo se encontro el 100";
35
36    // localiza la primera ocurrencia del valor que sea mayor que 10 en a
37    ubicacion = find_if( a.cbegin(), a.cend(), mayor10 );
38
39    if ( ubicacion != a.cend() ) // encontro un valor mayor que 10
40       cout << "\n\nEl primer valor mayor que 10 es " << *ubicacion
41          << "\nse encontro en la ubicacion " << ( ubicacion - a.cbegin() );
42    else // no se encontró un valor mayor que 10
43       cout << "\n\nNo se encontraron valores mayores que 10";
44
45    // ordena los elementos de a
46    sort( a.begin(), a.end() );
47    cout << "\n\narreglo a despues de sort: ";
48    copy( a.cbegin(), a.cend(), salida );
49
50    // usa binary_search para localizar el 13 en a
51    if ( binary_search( a.cbegin(), a.cend(), 13 ) )
52       cout << "\n\nSe encontro el 13 en a";
53    else
54       cout << "\n\nNo se encontro el 13 en a";
55
56    // usa binary_search para localizar el 100 en a
57    if ( binary_search( a.cbegin(), a.cend(), 100 ) )
58       cout << "\nSe encontro el 100 en a";
59    else
60       cout << "\nNo se encontro el 100 en a";
61
62    // determina si todos los elementos de a son mayores que 10
63    if ( all_of( a.cbegin(), a.cend(), mayor10 ) )
64       cout << "\n\nTodos los elementos en a son mayores que 10";
65    else
66       cout << "\n\nAlgunos elementos en a no son mayores que 10";
67
```

Fig. 16.6 | Algoritmos básicos de búsqueda y ordenamiento de la Biblioteca estándar (parte 2 de 3).

```
68    // determina si alguno de los elementos de a es mayor que 10
69    if ( any_of( a.cbegin(), a.cend(), mayor10 ) )
70       cout << "\n\nAlgunos de los elementos en a son mayores que 10";
71    else
72       cout << "\n\nNinguno de los elementos en a son mayores que 10";
73
74    // determina si ninguno de los elementos de a es mayor que 10
75    if ( none_of( a.cbegin(), a.cend(), mayor10 ) )
76       cout << "\n\nNinguno de los elementos en a es mayor que 10";
77    else
78       cout << "\n\nAlgunos de los elementos en a son mayores que 10";
79
80    // localiza la primera ocurrencia del valor que no sea mayor que 10 en a
81    ubicacion = find_if_not( a.cbegin(), a.cend(), mayor10 );
82
83    if ( ubicacion != a.cend() ) // encontró un valor menor o igual a 10
84       cout << "\n\nEl primer valor no mayor que 10 es " << *ubicacion
85          << "\nse encontro en la ubicacion " << ( ubicacion - a.cbegin() );
86    else // no se encontraron valores menores o iguales a 10
87       cout << "\n\nSolo se encontraron valores mayores que 10 ";
88
89    cout << endl;
90 } // fin de main
91
92 // determina si el argumento es mayor que 10
93 bool mayor10( int valor )
94 {
95    return valor > 10;
96 } // fin de la función mayor10
```

```
El arreglo a contiene: 10 2 17 5 16 8 13 11 20 7

Se encontro el 16 en la ubicacion 4
No se encontro el 100

El primer valor mayor que 10 es 17
se encontro en la ubicacion 2

arreglo a despues de sort: 2 5 7 8 10 11 13 16 17 20

Se encontro el 13 en a
No se encontro el 100 en a

Algunos elementos en a no son mayores que 10

Algunos de los elementos en a son mayores que 10

Algunos de los elementos en a son mayores que 10

El primer valor no mayor que 10 es 2
se encontro en la ubicación 0
```

Fig. 16.6 | Algoritmos básicos de búsqueda y ordenamiento de la Biblioteca estándar (parte 3 de 3).

Algoritmo `find`

En la línea 21 se utiliza la función **`find`** para localizar el valor 16 en el rango que empieza desde `a.cbegin()` hasta, pero sin incluir a, `a.cend()`. El algoritmo requiere que sus dos argumentos iteradores sean por lo

menos *iteradores de entrada* y devuelve un *iterador de entrada* que, o se coloca en el primer elemento que contiene el valor, o indica el final de la secuencia (como es el caso en la línea 29).

Algoritmo `find_if`

En la línea 37 se utiliza el algoritmo `find_if` (una búsqueda lineal) para localizar el primer valor en el rango empezando desde `a.cbegin()` hasta, pero *sin* incluir a, `a.cend()` para el que la *función predicado unaria* `mayor10` devuelva `true`. Esta función `mayor10` (definida en las líneas 93 a 96) toma un entero y devuelve un valor `bool` que indica si el argumento entero es *mayor que* 10. El algoritmo `find_if` requiere que sus dos argumentos iteradores sean por lo menos *iteradores de entrada*. El algoritmo devuelve un *iterador de entrada* que, o se coloca en el primer elemento que contiene un valor para el que la función predicado devuelva `true`, o indica el final de la secuencia.

Algoritmo `sort`

En la línea 46 se utiliza la función `sort` para ordenar los elementos en el rango que empieza desde `a.begin()` hasta, pero *sin* incluir a, `a.end()` en *orden ascendente*. El algoritmo requiere que sus dos argumentos iteradores sean *iteradores de acceso aleatorio*. Una segunda versión de este algoritmo toma un tercer argumento, el cual es una *función predicado binaria* que toma dos argumentos que son valores en la secuencia y devuelve un `bool` que indica el *orden* de los elementos; si el valor de retorno es `true`, los dos elementos que se están comparando se encuentran en *orden*.

Algoritmo `binary_search`

En la línea 55 se utiliza la función `binary_search` para determinar si el valor 13 se encuentra en el rango empezando desde `a.cbegin()` hasta, pero *sin* incluir a, `a.cend()`. Los valores debe ordenarse en *forma ascendente*. El algoritmo `binary_search` requiere que sus dos argumentos iteradores sean por lo menos *iteradores de avance*. El algoritmo devuelve un valor `bool` que indica si se encontró el valor en la secuencia. En la línea 57 se demuestra una llamada a la función `binary_search`, en donde el valor *no* se encontró. Una segunda versión de este algoritmo toma un cuarto argumento, el cual es una *función predicado binaria* que toma dos argumentos que son valores en la secuencia y devuelve un `bool`. La función predicado devuelve `true` si los dos elementos que se están comparando se encuentran en *orden*. Para obtener la *ubicación* de la clave de búsqueda en el contenedor, use los algoritmos `lower_bound` o `find`.

C++11: algoritmo `all_of`

En la línea 63 se utiliza el algoritmo `all_of` para determinar si la *función predicado unaria* `mayor10` devuelve `true` para *todos* los elementos en el rango desde `a.cbegin()` hasta, pero *sin* incluir a, `a.cend()`. El algoritmo `all_of` requiere que sus dos argumentos iteradores sean al menos *iteradores de entrada*.

C++11: algoritmo `any_of`

En la línea 69 se utiliza el algoritmo `any_of` para determinar si la *función predicado unaria* `mayor10` devuelve `true` para *al menos uno* de los elementos en el rango desde `a.cbegin()` hasta, pero *sin* incluir a, `a.cend()`. El algoritmo `any_of` requiere que sus dos argumentos iteradores sean al menos *iteradores de entrada*.

C++11: algoritmo `none_of`

En la línea 75 se utiliza el algoritmo `none_of` para determinar si la *función predicado unaria* `mayor10` devuelve `false` para *todos* los elementos en el rango desde `a.cbegin()` hasta, pero *sin* incluir a, `a.cend()`. El algoritmo `none_of` requiere que sus dos argumentos iteradores sean al menos *iteradores de entrada*.

C++11: algoritmo **`find_if_not`**

En la línea 81 se utiliza el algoritmo **`find_if_not`** para localizar el primer valor en el rango desde `a.cbegin()` hasta, pero *sin* incluir a, `a.cend()` para el que la *función predicado unaria* `mayor10` devuelva `false`. El algoritmo `find_if` requiere que sus dos argumentos iteradores sean al menos *iteradores de entrada*. El algoritmo devuelve un *iterador de entrada* que, o se posicione en el primer elemento que contenga un valor para el que la función predicado devuelva `false`, o indique el final de la secuencia.

16.3.7 `swap`, `iter_swap` y `swap_ranges`

La figura 16.7 demuestra el uso de los algoritmos `swap`, `iter_swap` y `swap_ranges` para *intercambiar* elementos.

```
1  // Fig. 16.7: fig16_07.cpp
2  // Los algoritmos iter_swap, swap y swap_ranges.
3  #include <iostream>
4  #include <array>
5  #include <algorithm> // definiciones de los algoritmos
6  #include <iterator>
7  using namespace std;
8
9  int main()
10 {
11    const size_t TAMANIO = 10;
12    array< int, TAMANIO > a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13    ostream_iterator< int > salida( cout, " " );
14
15    cout << "El arreglo a contiene:\n   ";
16    copy( a.cbegin(), a.cend(), salida ); // muestra el arreglo a
17
18    swap( a[ 0 ], a[ 1 ] ); // intercambia los elementos en las ubicaciones 0 y 1 de a
19
20    cout << "\nEl arreglo a despues de intercambiar a[0] y a[1] mediante swap:\n   ";
21    copy( a.cbegin(), a.cend(), salida ); // muestra el arreglo a
22
23    // usa iteradores para intercambiar los elementos en las ubicaciones 0 y 1 del
        arreglo a
24    iter_swap( a.begin(), a.begin() + 1 ); // intercambia con iteradores
25    cout << "\nEl arreglo a después de intercambiar a[0] y a[1] mediante
            iter_swap:\n   ";
26    copy( a.cbegin(), a.cend(), salida );
27
28    // intercambia los primeros cinco elementos del arreglo a con
29    // los últimos cinco elementos del arreglo a
30    swap_ranges( a.begin(), a.begin() + 5, a.begin() + 5  );
31
32    cout << "\nEl arreglo a despues de intercambiar los primeros cinco
             elementos\n"
33         << "con los ultimos cinco:\n   ";
34    copy( a.cbegin(), a.cend(), salida );
35    cout << endl;
36 } // fin de main
```

Fig. 16.7 | Los algoritmos `iter_swap`, `swap` y `swap_ranges` (parte 1 de 2).

```
El arreglo a contiene:
   1 2 3 4 5 6 7 8 9 10
El arreglo a despues de intercambiar a[0] y a[1] mediante swap:
   2 1 3 4 5 6 7 8 9 10
El arreglo a despues de intercambiar a[0] y a[1] mediante iter_swap:
   1 2 3 4 5 6 7 8 9 10
El arreglo a despues de intercambiar los primeros cinco elementos
con los ultimos cinco:
   6 7 8 9 10 1 2 3 4 5
```

Fig. 16.7 | Los algoritmos `iter_swap`, `swap` y `swap_ranges` (parte 2 de 2).

Algoritmo **`swap`**

En la línea 18 se utiliza el algoritmo **`swap`** para intercambiar dos valores. En este ejemplo se intercambian el primer y segundo elementos del arreglo `a`. La función toma como argumentos las referencias a los dos valores que se van a intercambiar.

Algoritmo **`iter_swap`**

En la línea 24 se utiliza el algoritmo **`iter_swap`** para intercambiar los dos elementos. La función toma dos *iteradores de avance* como argumentos (en este caso, apuntadores a elementos de un arreglo) e intercambia los valores en los elementos a los que hacen referencia los iteradores.

Algoritmo **`swap_ranges`**

En la línea 30 se utiliza la función **`swap_ranges`** para intercambiar los elementos en el rango que empieza desde `a.begin()` hasta, pero sin incluir a, `a.begin() + 5` con los elementos que empiezan desde la posición `a.begin() + 5`. La función requiere tres *iteradores de avance* como argumentos. Los primeros dos argumentos especifican el rango de elementos en la primera secuencia que va a intercambiarse con los elementos en la segunda secuencia, empezando a partir del iterador en el tercer argumento. En este ejemplo, las dos secuencias de valores se encuentran en el mismo arreglo, pero las secuencias pueden provenir de distintos arreglos o contenedores. Las secuencias no deben traslaparse. La secuencia de destino debe ser lo bastante grande como para contener todos los elementos de los rangos que se van a intercambiar.

16.3.8 `copy_backward`, `merge`, `unique` y `reverse`

La figura 16.8 demuestra el uso de los algoritmos `copy_backward`, `merge`, `unique` y `reverse`.

```
1  // Fig. 16.8: fig6_08.cpp
2  // Los algoritmos copy_backward, merge, unique y reverse.
3  #include <iostream>
4  #include <algorithm> // definiciones de los algoritmos
5  #include <array> // definición de la plantilla de clase array
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  int main()
10 {
11    const size_t TAMANIO = 5;
12    array< int, TAMANIO > a1 = { 1, 3, 5, 7, 9 };
```

Fig. 16.8 | Los algoritmos `copy_backward`, `merge`, `unique` y `reverse` (parte 1 de 2).

```
13     array< int, TAMANIO > a2 = { 2, 4, 5, 7, 9 };
14     ostream_iterator< int > salida( cout, " " );
15
16     cout << "El arreglo a1 contiene: ";
17     copy( a1.cbegin(), a1.cend(), salida ); // muestra a1
18     cout << "\nEl arreglo a2 contiene: ";
19     copy( a2.cbegin(), a2.cend(), salida ); // muestra a2
20
21     array< int, TAMANIO > resultados;
22
23     // coloca los elementos de a1 en resultados, en orden inverso
24     copy_backward( a1.cbegin(), a1.cend(), resultados.end() );
25     cout << "\n\nDespues de copy_backward, resultados contiene: ";
26     copy( resultados.cbegin(), resultados.cend(), salida );
27
28     array< int, TAMANIO + TAMANIO >  resultados2;
29
30     // combina los elementos de a1 y a2 en resultados2, en orden
31     merge( a1.cbegin(), a1.cend(), a2.cbegin(), a2.cend(),
32       resultados2.begin() );
33
34     cout << "\n\nDespues de combinar a1 y a2, resultados2 contiene: ";
35     copy( resultados2.cbegin(), resultados2.cend(), salida );
36
37     // elimina valores duplicados de resultados2
38     auto ubicacionFinal = unique( resultados2.begin(), resultados2.end() );
39
40     cout << "\n\nDespues de unique, resultados2 contiene: ";
41     copy( resultados2.begin(), ubicacionFinal, salida );
42
43     cout << "\n\nEl arreglo a1 despues de reverse: ";
44     reverse( a1.begin(), a1.end() ); // invierte los elementos de a1
45     copy( a1.cbegin(), a1.cend(), salida );
46     cout << endl;
47   } // fin de main
```

```
El arreglo a1 contiene: 1 3 5 7 9
El arreglo a2 contiene: 2 4 5 7 9

Despues de copy_backward, resultados contiene: 1 3 5 7 9

Despues de combinar a1 y a2, resultados2 contiene: 1 2 3 4 5 5 7 7 9 9

Despues de unique, resultados2 contiene: 1 2 3 4 5 7 9

El arreglo a1 despues de reverse: 9 7 5 3 1
```

Fig. 16.8 | Los algoritmos `copy_backward`, `merge`, `unique` y `reverse` (parte 2 de 2).

Algoritmo `copy_backward`

En la línea 24 se utiliza el algoritmo **`copy_backward`** para copiar elementos en el rango que empieza desde `a1.cbegin()` hasta, pero *sin* incluir a, `a1.cend()`, colocar los elementos en `resultados` empezando desde elemento que está antes de `resultados.end()` y avanzando hacia el inicio del `array`. El algoritmo devuelve un iterador colocado en el último elemento que se copia a `resultados` (es decir, el inicio de `resultados`, debido a la copia al revés). Los elementos se colocan en `resultados` en el mismo orden que `a1`. Este algoritmo requiere de tres *iteradores bidireccionales* como argumentos (iteradores que pue-

den *incrementarse* y *decrementarse* para iterar *hacia delante* y *hacia atrás* a través de una secuencia, respectivamente). Una diferencia entre `copy_backward` y `copy` es que el iterador que se devuelve de `copy` se coloca *después* del último elemento copiado, y el iterador que se devuelve de `copy_backward` se coloca *en* el último elemento copiado (es decir, el primer elemento en la secuencia). Además, `copy_backward` *puede* manipular rangos *traslapados* de elementos en un contenedor, siempre y cuando el primer elemento a copiar *no* se encuentre en el rango de destino de los elementos.

Además de los algoritmos `copy` y `copy_backward`, C++11 incluye ahora los algoritmos **`move`** y **`move_backward`**. Éstos utilizan la nueva semántica de movimiento de C++11 (que veremos en el capítulo 24, C++11: Additional Features) para mover, en vez de copiar, objetos de un contenedor a otro.

Algoritmo `merge`

En las líneas 31 y 32 se utiliza la función **`merge`** para combinar dos *secuencias* de valores *ordenados en forma ascendente* en una tercera secuencia ordenada también en forma ascendente. El algoritmo requiere de cinco iteradores como argumentos. Los primeros cuatro deben ser por lo menos *iteradores de entrada* y el último argumento debe ser por lo menos un *iterador de salida*. Los primeros dos argumentos especifican el rango de elementos en la primera secuencia ordenada (`a1`), los siguientes dos argumentos especifican el rango de elementos en la segunda secuencia ordenada (`a2`) y el último argumento especifica la posición inicial en la tercera secuencia (`resultados2`) en donde se van a mezclar los elementos. Una segunda versión de este algoritmo toma como su sexto argumento a una *función predicado binaria* que especifica la forma en que se van a *ordenar* los elementos.

Adaptadores de iteradores `back_inserter`, `front_inserter` *e* `inserter`

En la línea 28 se crea el arreglo `resultados2` con el número de elementos en `a1` y `a2`. Para utilizar el algoritmo `merge` se requiere que la secuencia en donde van a almacenarse los resultados sea por lo menos de un tamaño igual al de las dos secuencias que van a mezclarse. Si no desea asignar el número de elementos para la secuencia resultante antes de la operación `merge`, puede utilizar las siguientes instrucciones:

```
vector< int > resultados2;
merge( a1.begin(), a1.end(), a2.begin(), a2.end(),
  back_inserter( resultados2 ) );
```

El argumento `back_inserter( resultados2 )` utiliza la plantilla de función **`back_inserter`** (en el encabezado `<iterator>`) para el `vector resultados2`. Una función `back_inserter` llama a la función `push_back` predeterminada del contenedor para insertar un elemento al *final* del mismo. Si se inserta un elemento en un contenedor que no tenga más espacio disponible, *el contenedor aumenta su tamaño*; esta es la razón por la que usamos un `vector` en las instrucciones anteriores, ya que los contenedores `array` son de tamaño fijo. Por lo tanto, el número de elementos en el contenedor *no* tiene que conocerse de antemano. Hay otros dos insertadores: **`front_inserter`** (usa `push_front` para insertar un elemento al *inicio* de un contenedor especificado como su argumento) e **`inserter`** (usa `insert` para insertar un elemento *en* el iterador que se proporciona como su segundo argumento en el contenedor que se proporciona como su primer argumento).

Algoritmo `unique`

En la línea 38 se utiliza el algoritmo **`unique`** en la secuencia *ordenada* de elementos en el rango que empieza desde `resultados2.begin()` hasta, pero *sin* incluir a, `resultados2.end()`. Una vez que se aplica este algoritmo a una secuencia ordenada con valores *duplicados*, sólo se retiene *una* copia de cada valor en la secuencia. El algoritmo toma dos argumentos que deben ser por lo menos *iteradores de avance*. El algoritmo devuelve un iterador que se coloca *después del último elemento* en la secuencia de valores únicos. Los valores de todos los elementos en el contenedor después del último valor único están *indefinidos*.

Una segunda versión de esta función toma como tercer argumento a una *función predicado binaria* que especifica cómo comparar dos elementos para ver si son *iguales*.

Algoritmo `reverse`

En la línea 44 se utiliza el algoritmo **`reverse`** para invertir todos los elementos en el rango que empieza desde `a1.begin()` hasta, pero *sin* incluir a, `a1.end()`. El algoritmo toma dos argumentos que deben ser por lo menos *iteradores bidireccionales*.

C++11: algoritmos `copy_if` *y* `copy_n`

Ahora C++11 incluye los nuevos algoritmos de copia `copy_if` y `copy_n`. El algoritmo **`copy_if`** copia cada elemento de un rango si la *función predicado unaria* en su cuarto argumento devuelve `true` para ese elemento. Los iteradores que se suministran como los primeros dos argumentos deben ser *iteradores de entrada*. El iterador que se suministra como el tercer argumento debe ser un *iterador de salida*, de modo que el elemento que se va a copiar pueda *asignarse* a la ubicación de copia. Este algoritmo devuelve un iterador posicionado después del **último** elemento copiado.

El algoritmo **`copy_n`** copia el número de elementos especificados por su segundo argumento, desde la ubicación especificada por su primer argumento (un *iterador de entrada*). Los elementos se envían a la ubicación especificada por su tercer argumento (un *iterador de salida*).

16.3.9 `inplace_merge`, `unique_copy` y `reverse_copy`

La figura 16.9 demuestra el uso de los algoritmos `inplace_merge`, `unique_copy` y `reverse_copy`.

```
1  // Fig. 16.9: fig16_09.cpp
2  // Los algoritmos inplace_merge, reverse_copy y unique_copy.
3  #include <iostream>
4  #include <algorithm> // definiciones de los algoritmos
5  #include <array> // definición de la plantilla de clase array
6  #include <vector> // definición de la plantilla de clase vector
7  #include <iterator> // definición de back_inserter
8  using namespace std;
9
10 int main()
11 {
12    const int TAMANIO = 10;
13    array< int, TAMANIO > a1 = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
14    ostream_iterator< int > salida( cout, " " );
15
16    cout << "El arreglo a1 contiene: ";
17    copy( a1.cbegin(), a1.cend(), salida );
18
19    // combina la primera mitad de a1 con la segunda mitad de a1, de tal forma
20    // que a1 contiene un conjunto ordenado de elementos después de la combinación
21    inplace_merge( a1.begin(), a1.begin() + 5, a1.end() );
22
23    cout << "\nDespues de inplace_merge, a1 contiene: ";
24    copy( a1.cbegin(), a1.cend(), salida );
```

Fig. 16.9 | Los algoritmos `inplace_merge`, `unique_copy` y `reverse_copy` (parte 1 de 2).

```
25
26     vector< int > resultados1;
27
28     // copia sólo los elementos únicos de a1 a resultados1
29     unique_copy( a1.cbegin(), a1.cend(), back_inserter( resultados1 ) );
30     cout << "\nDespues de unique_copy, resultados1 contiene: ";
31     copy( resultados1.cbegin(), resultados1.cend(), salida );
32
33     vector< int > resultados2;
34
35     // copia los elementos de a1 a resultados2 en orden inverso
36     reverse_copy( a1.cbegin(), a1.cend(), back_inserter( resultados2 ) );
37     cout << "\nDespues de reverse_copy, resultados2 contiene: ";
38     copy( resultados2.cbegin(), resultados2.cend(), salida );
39     cout << endl;
40   } // fin de main
```

```
El arreglo a1 contiene: 1 3 5 7 9 1 3 5 7 9
Despues de inplace_merge, a1 contiene: 1 1 3 3 5 5 7 7 9 9
Despues de unique_copy, resultados1 contiene: 1 3 5 7 9
Despues de reverse_copy, resultados2 contiene: 9 9 7 7 5 5 3 3 1 1
```

Fig. 16.9 | Los algoritmos `inplace_merge`, `unique_copy` y `reverse_copy` (parte 2 de 2).

Algoritmo `inplace_merge`

La línea 21 utiliza el algoritmo **`inplace_merge`** para mezclar dos *secuencias ordenadas* de elementos en el mismo contenedor. En este ejemplo, los elementos de `a1.begin()` hasta, pero *sin* incluir a, `a1.begin() + 5` se mezclan con los elementos de `a1.begin() + 5` hasta, pero *sin* incluir a, `a1.end()`. Este algoritmo requiere que sus tres argumentos iteradores sean por lo menos *iteradores bidireccionales*. Una segunda versión de esta función toma como cuarto argumento a una *función predicado binaria* para comparar elementos en las dos secuencias.

Algoritmo `unique_copy`

En la línea 29 se utiliza el algoritmo **`unique_copy`** para crear una copia de todos los elementos únicos en la secuencia ordenada de valores, empezando desde `a1.cbegin()` hasta, pero *sin* incluir a, `a1.cend()`. Los elementos copiados se colocan en el `vector resultados1`. Los primeros dos argumentos deben ser por lo menos *iteradores de entrada* y el último argumento debe ser por lo menos un *iterador de salida*. En este ejemplo *no* asignamos previamente suficientes elementos en `resultados1` como para almacenar *todos* los elementos copiados de `a1`. Lo que hicimos fue utilizar la función `back_inserter` (definida en el encabezado `<iterator>`) para agregar elementos al final de `resultados1`. Esta función utiliza la función miembro `push_back` de `vector` para insertar elementos al final del `vector`. Como `back_inserter` *inserta* un elemento *en vez de reemplazar* el valor de un elemento existente, el `vector` puede crecer para dar cabida a más elementos. Una segunda versión del algoritmo `unique_copy` toma como cuarto argumento a una *función predicado binaria* para comparar la *igualdad* entre los elementos.

Algoritmo `reverse_copy`

En la línea 36 se utiliza el algoritmo **`reverse_copy`** para crear una copia inversa de los elementos en el rango que empieza desde `a1.cbegin()` hasta, pero *sin* incluir a, `a1.cend()`. Los elementos copiados se insertan en `resultados2` mediante el uso de un objeto `back_inserter` para asegurar que el `vector`

pueda *crecer* para dar cabida al número apropiado de elementos que se copien. El algoritmo `reverse_copy` requiere que sus primeros dos argumentos iteradores sean por lo menos *iteradores bidireccionales* y que su tercer argumento iterador sea por lo menos un *iterador de salida*.

16.3.10 Operaciones establecer (set)

En la figura 16.10 se demuestra el uso de los algoritmos `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` y `set_union` para manipular *conjuntos de valores ordenados*.

```
1  // Fig. 16.10: fig16_10.cpp
2  // Los algoritmos includes, set_difference, set_intersection,
3  // set_symmetric_difference y set_union.
4  #include <iostream>
5  #include <array>
6  #include <algorithm> // definiciones de los algoritmos
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 int main()
11 {
12    const size_t TAMANIO1 = 10, TAMANIO2 = 5, TAMANIO3 = 20;
13    array< int, TAMANIO1 > a1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14    array< int, TAMANIO2 > a2 = { 4, 5, 6, 7, 8 };
15    array< int, TAMANIO2 > a3 = { 4, 5, 6, 11, 15 };
16    ostream_iterator< int > salida( cout, " " );
17
18    cout << "a1 contiene: ";
19    copy( a1.cbegin(), a1.cend(), salida ); // muestra el arreglo a1
20    cout << "\na2 contiene: ";
21    copy( a2.cbegin(), a2.cend(), salida ); // muestra el arreglo a2
22    cout << "\na3 contiene: ";
23    copy( a3.cbegin(), a3.cend(), salida ); // muestra el arreglo a3
24
25    // determina si a2 está completamente contenido en a1
26    if ( includes( a1.cbegin(), a1.cend(), a2.cbegin(), a2.cend() ) )
27       cout << "\n\na1 incluye a a2";
28    else
29       cout << "\n\na1 no incluye a a2";
30
31    // determina si a3 está completamente contenido en a1
32    if ( includes( a1.cbegin(), a1.cend(), a3.cbegin(), a3.cend() ) )
33       cout << "\na1 incluye a a3";
34    else
35       cout << "\na1 no incluye a a3";
36
37    array< int, TAMANIO1 > diferencia;
38
39    // determina los elementos de a1 que no están en a2
40    auto resultado1 = set_difference( a1.cbegin(), a1.cend(),
41       a2.cbegin(), a2.cend(), diferencia.begin() );
```

Fig. 16.10 | Los algoritmos `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` y `set_union` (parte 1 de 2).

```
42     cout << "\n\nset_difference de a1 y a2 es: ";
43     copy( diferencia.begin(), resultado1, salida );
44
45     array< int, TAMANIO1 > interseccion;
46
47     // determina los elementos que están tanto en a1 como en a2
48     auto resultado2 = set_intersection( a1.cbegin(), a1.cend(),
49        a2.cbegin(), a2.cend(), interseccion.begin() );
50     cout << "\n\nset_intersection de a1 y a2 es: ";
51     copy( interseccion.begin(), resultado2, salida );
52
53     array< int, TAMANIO1 + TAMANIO2 > symmetric_difference;
54
55     // determina los elementos de a1 que no están en a2 y
56     // los elementos de a2 que no están en a1
57     auto resultado3 = set_symmetric_difference( a1.cbegin(), a1.cend(),
58        a3.cbegin(), a3.cend(), symmetric_difference.begin() );
59     cout << "\n\nset_symmetric_difference de a1 y a3 es: ";
60     copy( symmetric_difference.begin(), resultado3, salida );
61
62     array< int, TAMANIO3 > conjuntoUnion;
63
64     // determina los elementos que están en uno o ambos conjuntos
65     auto resultado4 = set_union( a1.cbegin(), a1.cend(),
66        a3.cbegin(), a3.cend(), conjuntoUnion.begin() );
67     cout << "\n\nset_union de a1 y a3 es: ";
68     copy( conjuntoUnion.begin(), resultado4, salida );
69     cout << endl;
70  } // fin de main
```

```
a1 contiene: 1 2 3 4 5 6 7 8 9 10
a2 contiene: 4 5 6 7 8
a3 contiene: 4 5 6 11 15

a1 incluye a a2
a1 no incluye a a3

set_difference de a1 y a2 es: 1 2 3 9 10

set_intersection de a1 y a2 es: 4 5 6 7 8

set_symmetric_difference de a1 y a3 es: 1 2 3 7 8 9 10 11 15

set_union de a1 y a3 es: 1 2 3 4 5 6 7 8 9 10 11 15
```

Fig. 16.10 | Los algoritmos `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` y `set_union` (parte 2 de 2).

Algoritmo `includes`

En las líneas 26 y 32 se llama al algoritmo **`includes`**, que compara dos conjuntos de valores *ordenados* para determinar si *cada* elemento del segundo conjunto se encuentra en el primero. De ser así, `includes` devuelve `true`; en caso contrario, devuelve `false`. Los primeros dos argumentos iteradores deben ser por lo menos *iteradores de entrada* y deben describir el primer conjunto de valores. En la línea 26, el primer conjunto consiste de los elementos desde `a1.cbegin()` hasta, pero *sin* incluir a, `a1.cend()`. Los últimos dos argumentos iteradores deben ser por lo menos *iteradores de entrada* y deben describir el segundo conjunto de valores. En este ejemplo, el segundo conjunto consiste de los elementos desde

`a2.cbegin()` hasta, pero *sin* incluir a, `a2.cend()`. Una segunda versión del algoritmo `includes` toma un quinto argumento, el cual es una *función predicado binaria* que indica el orden en el que se ordenaron originalmente los elementos. Las dos secuencias deben ordenarse mediante el uso de la *misma función de comparación*.

Algoritmo `set_difference`

En las líneas 40 y 41 se utiliza el algoritmo `set_difference` para buscar los elementos del primer conjunto de valores ordenados que *no* se encuentren en el segundo conjunto de valores ordenados (ambos conjuntos de valores deben estar en *orden ascendente*). Los elementos que son *diferentes* se copian en el quinto argumento (en este caso, en el arreglo `diferencia`). Los primeros dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un *iterador de salida* que indique en dónde se debe almacenar una copia de los valores que sean *distintos*. El algoritmo devuelve un *iterador de salida* que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de la función `set_difference` toma un sexto argumento que viene siendo una *función predicado binaria*, la cual indica el *orden* en el que se *encontraban originalmente* los elementos. Las dos secuencias deben ordenarse mediante la *misma función de comparación*.

Algoritmo `set_intersection`

En las líneas 48 y 49 se utiliza el algoritmo `set_intersection` para determinar qué elementos del primer conjunto de valores ordenados se *encuentran* en el segundo conjunto de valores ordenados (ambos conjuntos de valores deben estar en orden *ascendente*). Los elementos *comunes en ambos conjuntos* se copian al quinto argumento (en este caso, en el arreglo `interseccion`). Los primeros dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un *iterador de salida* que indique en dónde se debe almacenar una copia de los valores que sean iguales. El algoritmo devuelve un *iterador de salida* que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de `set_intersection` toma un sexto argumento que viene siendo una *función predicado binaria*, la cual indica el orden en el que se encontraban *originalmente* los elementos. Las dos secuencias deben ordenarse mediante la *misma función de comparación*.

Algoritmo `set_symmetric_difference`

En las líneas 57 y 58 se utiliza el algoritmo `set_symmetric_difference` para determinar qué elementos en el primer conjunto *no* se encuentran en el segundo, y qué elementos en el segundo conjunto *no* se encuentran en el primero (ambos conjuntos deben estar en *orden ascendente*). Los elementos que sean *diferentes* se copian de ambos conjuntos hacia el quinto argumento (el arreglo `symmetric_difference`). Los primeros dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un *iterador de salida* que indique en dónde debe almacenarse una copia de los valores que sean diferentes. El algoritmo devuelve un *iterador de salida* que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de la función `set_symmetric_difference` toma un sexto argumento que viene siendo una *función predicado binaria*, la cual indica el orden en el que se encontraban originalmente los elementos. Las dos secuencias deben ordenarse mediante la *misma función de comparación*.

Algoritmo **`set_union`**

En las líneas 65 y 66 se utiliza el algoritmo **`set_union`** para crear un conjunto de todos los elementos que se encuentran en *cada uno* de los dos conjuntos ordenados, o *en ambos* (los dos conjuntos de valores deben estar en *orden ascendente*). Los elementos se copian de ambos conjuntos hacia el quinto argumento (en este caso, el arreglo `conjuntoUnion`). Los elementos que aparecen en *ambos* conjuntos sólo se copian del primer conjunto. Los primeros dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos *iteradores de entrada* para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un *iterador de salida* que indique en dónde deben almacenarse los elementos copiados. El algoritmo devuelve un *iterador de salida* que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de `set_union` toma un sexto argumento que viene siendo una *función predicado binaria*, la cual indica el orden en el que se encontraban *originalmente* los elementos. Las dos secuencias deben ordenarse mediante la *misma función de comparación*.

16.3.11 `lower_bound`, `upper_bound` y `equal_range`

En la figura 16.11 se demuestra el uso de los algoritmos `lower_bound`, `upper_bound` y `equal_range`.

```
1  // Fig. 16.11: fig16_11.cpp
2  // Los algoritmos lower_bound, upper_bound y
3  // equal_range para una secuencia ordenada de valores.
4  #include <iostream>
5  #include <algorithm> // definiciones de los algoritmos
6  #include <array> // definición de la plantilla de clase array
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 int main()
11 {
12    const size_t TAMANIO = 10;
13    array< int, TAMANIO > a = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
14    ostream_iterator< int > salida( cout, " " );
15
16    cout << "El arreglo a contiene:\n";
17    copy( a.cbegin(), a.cend(), salida );
18
19    // determina el punto de inserción del límite inferior para 6 en a
20    auto inferior = lower_bound( a.cbegin(), a.cend(), 6 );
21    cout << "\n\nEl limite inferior de 6 es el elemento "
22       << ( inferior - a.cbegin() ) << " del vector v";
23
24    // determina el punto de inserción del límite superior para 6 en a
25    auto superior = upper_bound( a.cbegin(), a.cend(), 6 );
26    cout << "\nEl limite superior de 6 es el elemento "
27       << ( superior - a.cbegin() ) << " del arreglo a ";
28
29    // usa equal_range para determinar los puntos de inserción
30    // inferior y superior para 6
31    auto eq = equal_range( a.cbegin(), a.cend(), 6 );
```

Fig. 16.11 | Los algoritmos `lower_bound`, `upper_bound` y `equal_range` para una secuencia ordenada de valores (parte 1 de 2).

```
32     cout << "\nUsando equal_range:\n   El limite inferior de 6 es el elemento "
33          << ( eq.first - a.cbegin() ) << " del arreglo a";
34     cout << "\n   El limite superior de 6 es el elemento "
35       << ( eq.second - a.cbegin() ) << " del arreglo a";;
36     cout << "\n\nUsa lower_bound para localizar el primer punto\n"
37       << "en el que se puede insertar el 5 en orden";
38
39     // determina el punto de inserción del límite inferior para 5 en a
40     inferior = lower_bound( a.cbegin(), a.cend(), 5 );
41     cout << "\n   El limite inferior de 5 es el elemento "
42       << ( inferior - a.cbegin() ) << " del arreglo a";
43     cout << "\n\nUsa upper_bound para localizar el ultimo punto\n"
44       << "en el que se puede insertar el 7 en orden";
45
46     // determina el punto de inserción del límite superior para 7 en a
47     superior = upper_bound( a.cbegin(), a.cend(), 7 );
48     cout << "\n   El limite superior de 7 es el elemento "
49       << ( superior - a.cbegin() ) << " del arreglo a";
50     cout << "\n\nUsa equal_range para localizar el primer y\n"
51       << "ultimo punto en el que se puede insertar el 5 en orden";
52
53     // usa equal_range para determinar los puntos de inserción
54     // inferior y superior para el 5
55     eq = equal_range( a.cbegin(), a.cend(), 5 );
56     cout << "\n   El limite inferior de 5 es el elemento "
57       << ( eq.first - a.cbegin() ) << " del arreglo a";
58     cout << "\n   El limite superior de 5 es el elemento "
59       << ( eq.second - a.cbegin() ) << " del arreglo a" << endl;
60   } // fin de main
```

```
El arreglo a contiene:
2 2 4 4 4 6 6 6 6 8

El limite inferior de 6 es el elemento 5 del arreglo a
El limite superior de 6 es el elemento 9 del arreglo a
Usando equal_range:
   El limite inferior de 6 es el elemento 5 del arreglo a
   El limite superior de 6 es el elemento 9 del arreglo a

Usa lower_bound para localizar el primer punto
en el que se puede insertar el 5 en orden
   El limite inferior de 5 es el elemento 5 del arreglo a

Usa upper_bound para localizar el ultimo punto
en el que se puede insertar el 7 en orden
   El limite superior de 7 es el elemento 9 del arreglo a

Usa equal_range para localizar el primer y
ultimo punto en el que se puede insertar el 5 en orden
   El limite inferior de 5 es el elemento 5 del arreglo a
   El limite superior de 5 es el elemento 5 del arreglo a
```

Fig. 16.11 | Los algoritmos `lower_bound`, `upper_bound` y `equal_range` para una secuencia ordenada de valores (parte 2 de 2).

Algoritmo `lower_bound`

En la línea 20 se utiliza el algoritmo **`lower_bound`** para buscar la primera posición en una secuencia ordenada de valores en donde pueda insertarse el tercer argumento, de manera que la secuencia permanezca en *orden ascendente*. Los primeros dos argumentos iteradores deben ser por lo menos *iteradores de avance*. El tercer argumento es el valor para el que se debe determinar el límite inferior. El algoritmo devuelve un *iterador de avance* que apunta a la posición en la que puede realizarse la inserción. Una segunda versión de `lower_bound` toma como cuarto argumento una *función predicado binaria*, la cual indica el orden en el que se encontraban *originalmente* los elementos.

Algoritmo `upper_bound`

En la línea 25 se utiliza el algoritmo **`upper_bound`** para buscar la última posición en una secuencia ordenada de valores en donde pueda insertarse el tercer argumento, de manera que la secuencia permanezca en *orden ascendente*. Los primeros dos argumentos iteradores deben ser por lo menos *iteradores de avance*. El tercer argumento es el valor para el que se va a determinar el límite superior. El algoritmo devuelve un *iterador de avance* que apunta a la posición en la que puede realizarse la inserción. Una segunda versión de `upper_bound` toma como cuarto argumento una *función predicado binaria*, la cual indica el orden en el que se encontraban *originalmente* los elementos.

Algoritmo `equal_range`

En la línea 31 se utiliza el algoritmo **`equal_range`** para devolver un par (un objeto **`pair`**) de *iteradores de avance* que contienen los resultados de llevar a cabo las operaciones `lower_bound` y `upper_bound`. Los primeros dos argumentos deben ser por lo menos *iteradores de avance*. El tercer argumento es el valor para el que se va a localizar el rango equivalente. El algoritmo devuelve un par de *iteradores de avance* para los límites inferior (`eq.first`) y superior (`eq.second`), respectivamente.

Localización de puntos de inserción en secuencias ordenadas

Las funciones `lower_bound`, `upper_bound` e `equal_range` se utilizan a menudo para localizar *puntos de inserción* en secuencias ordenadas. En la línea 40 se utiliza `lower_bound` para localizar el primer punto en el que puede insertarse un `5` en orden, en `a`. En la línea 47 se utiliza `upper_bound` para localizar el último punto en el que puede insertarse un `7` en orden, en `a`. En la línea 55 se utiliza `equal_range` para localizar el primer y último puntos en los que puede insertarse un `5` en orden, en `a`.

16.3.12 Ordenamiento de montón (heapsort)

En la figura 16.12 se demuestra el uso de los algoritmos de la Biblioteca estándar para llevar a cabo el **algoritmo de ordenamiento heapsort**, mediante el cual se ordena un arreglo de elementos en una estructura de datos conocida como *montón (heap)*. El algoritmo heapsort se describe con detalle en los cursos de ciencias computacionales llamados "Estructuras de datos" y "Algoritmos". Si desea más información y recursos adicionales, visite:

```
en.wikipedia.org/wiki/Heapsort
```

```
1  // Fig. 16.12: fig16_12.cpp
2  // Los algoritmos push_heap, pop_heap, make_heap y sort_heap.
3  #include <iostream>
4  #include <algorithm>
5  #include <array>
```

Fig. 16.12 | Los algoritmos `push_heap`, `pop_heap`, `make_heap` y `sort_heap` (parte 1 de 3).

```
#include <vector>
#include <iterator>
using namespace std;

int main()
{
   const size_t TAMANIO = 10;
   array< int, TAMANIO > inic = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
   array< int, TAMANIO > a( inic ); // copia de inic
   ostream_iterator< int > salida( cout, " " );

   cout << "El arreglo a antes de make_heap:\n";
   copy( a.cbegin(), a.cend(), salida );

   make_heap( a.begin(), a.end() ); // crea un montón en base al arreglo a
   cout << "\nEl arreglo a despues de make_heap:\n";
   copy( a.cbegin(), a.cend(), salida );

   sort_heap( a.begin(), a.end() ); // ordena los elementos con sort_heap
   cout << "\nEl vector a despues de sort_heap:\n";
   copy( a.cbegin(), a.cend(), salida );

   // realiza el algoritmo heapsort con push_heap y pop_heap
   cout << "\n\nEl arreglo inic contiene: ";
   copy( inic.cbegin(), inic.cend(), salida ); // muestra el arreglo inic
   cout << endl;

   vector< int > v;

   // coloca los elementos del arreglo inic en v y
   // mantiene los elementos de v en el montón
   for ( size_t i = 0; i < TAMANIO; ++i )
   {
      v.push_back( inic[ i ] );
      push_heap( v.begin(), v.end() );
      cout << "\nv despues de push_heap(inic[" << i << "]): ";
      copy( v.cbegin(), v.cend(), salida );
   } // fin de for

   cout << endl;

   // elimina los elementos del montón en orden
   for ( size_t  j = 0; j < v.size(); ++j )
   {
      cout << "\nv despues de sacar " << v[ 0 ] << " del monton\n";
      pop_heap( v.begin(), v.end() - j );
      copy( v.cbegin(), v.cend(), salida );
   } // fin de for

   cout << endl;
} // fin de main
```

Fig. 16.12 | Los algoritmos `push_heap`, `pop_heap`, `make_heap` y `sort_heap` (parte 2 de 3).

```
El arreglo a antes de make_heap:
3 100 52 77 22 31 1 98 13 40
El arreglo a despues de make_heap:
100 98 52 77 40 31 1 3 13 22
El arreglo a despues de sort_heap:
1 3 13 22 31 40 52 77 98 100

El arreglo inic contiene: 3 100 52 77 22 31 1 98 13 40

v despues de push_heap(inic[0]): 3
v despues de push_heap(inic[1]): 100 3
v despues de push_heap(inic[2]): 100 3 52
v despues de push_heap(inic[3]): 100 77 52 3
v despues de push_heap(inic[4]): 100 77 52 3 22
v despues de push_heap(inic[5]): 100 77 52 3 22 31
v despues de push_heap(inic[6]): 100 77 52 3 22 31 1
v despues de push_heap(inic[7]): 100 98 52 77 22 31 1 3
v despues de push_heap(inic[8]): 100 98 52 77 22 31 1 3 13
v despues de push_heap(inic[9]): 100 98 52 77 40 31 1 3 13 22

v despues de sacar 100 del monton
98 77 52 22 40 31 1 3 13 100
v despues de sacar 98 del monton
77 40 52 22 13 31 1 3 98 100
v despues de sacar 77 del monton
52 40 31 22 13 3 1 77 98 100
v despues de sacar 52 del monton
40 22 31 1 13 3 52 77 98 100
v despues de sacar 40 del monton
31 22 3 1 13 40 52 77 98 100
v despues de sacar 31 del monton
22 13 3 1 31 40 52 77 98 100
v despues de sacar 22 del monton
13 1 3 22 31 40 52 77 98 100
v despues de sacar 13 del monton
3 1 13 22 31 40 52 77 98 100
v despues de sacar 3 del monton
1 3 13 22 31 40 52 77 98 100
v despues de sacar 1 del monton
1 3 13 22 31 40 52 77 98 100
```

Fig. 16.12 | Los algoritmos `push_heap`, `pop_heap`, `make_heap` y `sort_heap` (parte 3 de 3).

Algoritmo `make_heap`

En la línea 20 se utiliza la función **`make_heap`** para tomar una secuencia de valores en el rango que empieza desde `a.begin()` hasta, pero *sin* incluir a, `a.end()`, y *crear un montón* que pueda utilizarse para producir una *secuencia ordenada*. Los dos iteradores que se toman como argumentos deben ser *iteradores de acceso aleatorio*, por lo que este algoritmo sólo trabaja con contenedores `array`, `vector` y `deque`. Una segunda versión de este algoritmo toma como tercer argumento a una *función predicado binaria* para *comparar* valores.

Algoritmo `sort_heap`

En la línea 24 se utiliza el algoritmo **`sort_heap`** para *ordenar una secuencia de valores* en el rango que empieza desde `a.begin()` hasta, pero *sin* incluir a, `a.end()`, que ya se encuentran ordenados en un montón. Los dos iteradores que se toman como argumentos deben ser *iteradores de acceso aleatorio*. Una segunda versión de este argumento toma como tercer argumento a una *función predicado binaria* para *comparar* valores.

Algoritmo **push_heap**

En la línea 40 se utiliza la función **push_heap** para *agregar un nuevo valor a un montón*. Tomamos un elemento del arreglo `inic` a la vez, *anexando ese elemento* al *final* del `vector` `v` y realizamos la operación `push_heap`. Si el elemento anexado es el único en el `vector`, éste es *ya* un montón. De no ser así, `push_heap` reordena los elementos del `vector` en un montón. Cada vez que se llama a `push_heap`, se asume que el último elemento actualmente en el `vector` (es decir, el que se anexa antes de la llamada a `push_heap`) es el que se va a agregar al montón, y que todos los demás elementos en el `vector` ya se encuentran ordenados como un montón. Los dos argumentos iteradores para `push_heap` deben ser *iteradores de acceso aleatorio*. Una segunda versión de este algoritmo toma como tercer argumento a una *función predicado binaria* para *comparar* valores.

Algoritmo **pop_heap**

En la línea 51 se utiliza **pop_heap** para eliminar el elemento de la *parte superior* del montón. Este algoritmo asume que los elementos en el rango especificado por sus dos argumentos *iteradores de acceso aleatorio* ya son un montón. El proceso de eliminar en forma repetida el elemento *superior* del montón produce una secuencia ordenada de valores. El algoritmo `pop_heap` *intercambia* el *primer* elemento del montón (`v.begin()`) con su último elemento (el elemento antes de `v.end() - j`) y luego se asegura que los elementos hasta (pero *sin* incluir) el último elemento sigan formando un montón. Observe en la salida que, después de las operaciones con `pop_heap`, el `vector` está *ordenado* en *forma ascendente*. Una segunda versión de este algoritmo toma como tercer argumento a una *función predicado binaria* para comparar valores.

C++11: algoritmos **is_heap** *e* **is_heap_until**

Además de los algoritmos `make_heap`, `sort_heap`, `push_heap` y `pop_heap` que presentamos en la figura 16.12, ahora C++11 incluye los nuevos algoritmos `is_heap` e `is_heap_until`. El algoritmo **is_heap** devuelve `true` si los elementos en el rango especificado representan un montón. Una segunda versión de este algoritmo toma como tercer argumento una *función predicado binaria* para comparar los valores.

El algoritmo `is_heap_until` comprueba el rango especificado de valores y devuelve un iterador que apunta al último elemento en el rango para el que los elementos hasta (pero sin incluir a) ese iterador representan un montón.

16.3.13 `min`, `max`, `minmax` y `minmax_element`

En la figura 16.13 se demuestra el uso de los algoritmos `min`, `max`, `minmax` y `minmax_element`.

```
1  // Fig. 16.13: fig16_13.cpp
2  // Los algoritmos min, max, minmax y minmax_element.
3  #include <iostream>
4  #include <array>
5  #include <algorithm>
6  using namespace std;
7
8  int main()
9  {
10    cout << "El minimo de 12 y 7 es: " << min( 12, 7 );
11    cout << "\nEl maximo de 12 y 7 es: " << max( 12, 7 );
12    cout << "\nEl minimo de 'G' y 'Z' es: " << min( 'G', 'Z' );
```

Fig. 16.13 | Los algoritmos `min`, `max`, `minmax` y `minmax_element` (parte 1 de 2).

```
13   cout << "\nEl maximo de 'G' y 'Z' es: " << max( 'G', 'Z' );
14
15      // determina cuál argumento es el mínimo y cuál es el máximo
16      auto resultado1 = minmax( 12, 7 );
17      cout << "\n\nEl minimo de 12 y 7 es: " << resultado1.first
18         << "\nEl maximo de 12 y 7 es: " << resultado1.second;
19
20      array< int, 10 > elementos = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
21      ostream_iterator< int > salida( cout, " " );
22
23      cout << "\n\nEl arreglo elementos contiene: ";
24      copy( elementos.cbegin(), elementos.cend(), salida );
25
26      auto resultado2 = minmax_element( elementos.cbegin(), elementos.cend() );
27      cout << "\nEl elemento minimo en elementos es: " << *resultado2.first
28         << "\nEl elemento maximo en elementos es: " << *resultado2.second
29         << endl;
30   } // fin de main
```

```
El minimo de 12 y 7 es: 7
El maximo de 12 y 7 es: 12
El minimo de 'G' y 'Z' es: G
El maximo de 'G' y 'Z' es: Z

El minimo de 12 y 7 es: 7
El maximo de 12 y 7 es: 12

El arreglo elementos contiene: 3 100 52 77 22 31 1 98 13 40
El elemento minimo en elementos es: 1
El elemento maximo en elementos es: 100
```

Fig. 16.13 | Los algoritmos `min`, `max`, `minmax` y `minmax_element` (parte 2 de 2).

Algoritmos `min` *y* `max` *con dos parámetros*

Los algoritmos **`min`** y **`max`** (que se demuestran en las líneas 10 a 13) determinan el mínimo y el máximo de dos elementos, respectivamente.

C++11: algoritmos `min` *y* `max` *con parámetros* `initializer_list`

Ahora C++11 incluye versiones sobrecargadas de los algoritmos `min` y `max`, cada uno de los cuales recibe un parámetro `initializer_list` y devuelve el elemento menor o mayor en el inicializador de lista que se pasa como argumento. Por ejemplo, la siguiente instrucción devuelve 7:

```
int minimo = min( { 10, 7, 14, 21, 17 } );
```

Cada uno de estos nuevos algoritmos `min` y `max` está sobrecargado con una versión que toma como segundo argumento una *función predicado binaria* para comparar valores.

C++11: algoritmo `minmax`

Ahora C++11 incluye el algoritmo **`minmax`** (línea 16) que recibe dos elementos y devuelve un `pair` en donde el elemento más pequeño se almacena en `first` y el elemento más grande se almacena en `second`. Una segunda versión de este algoritmo toma como tercer argumento una *función predicado binaria* para comparar valores.

C++11: algoritmo **`minmax_element`**
Ahora C++11 incluye el algoritmo **`minmax_element`** (línea 26) que recibe dos *iteradores de entrada* que representan un rango de elementos, y devuelve un `pair` de iteradores en donde `first` apunta al elemento más pequeño en el rango y `second` apunta al más grande. Una segunda versión de este algoritmo toma como tercer argumento una *función predicado binaria* para comparar valores.

16.4 Objetos de funciones

Muchos algoritmos de la Biblioteca estándar nos permiten pasar un *apuntador a una función* al algoritmo, para ayudarlo a que lleve a cabo su tarea. Por ejemplo, el algoritmo `binary_search` que vimos en la sección 16.3.6 está sobrecargado con una versión que requiere como su cuarto parámetro un *apuntador a una función* que recibe dos argumentos y devuelve un valor `bool`. El algoritmo utiliza esta función para comparar la clave de búsqueda con un elemento en la colección. La función devuelve `true` si la clave de búsqueda y el elemento que se van a comparar son iguales; en caso contrario, la función devuelve `false`. Esto permite a `binary_search` buscar en una colección de elementos para la cual el tipo del elemento *no* proporcione un operador de igualdad < sobrecargado.

Cualquier algoritmo que puede recibir un *apuntador a una función* también puede recibir un objeto de una clase que sobrecargue al operador de llamada a función (paréntesis) con una función llamada `operator()`, siempre y cuando el operador sobrecargado cumpla con los requerimientos del algoritmo; en el caso de `binary_search`, debe recibir dos argumentos y devolver un valor `bool`. Un objeto de dicha clase se conoce como **objeto de función**, y se puede usar en forma sintáctica y semántica como una función o un *apuntador a una función*; el operador paréntesis sobrecargado se invoca mediante el uso del nombre de un objeto de función, seguido por paréntesis que contienen los argumentos para la función. La mayoría de los algoritmos pueden utilizar objetos de función y funciones de igual forma. Como veremos en la sección 16.5, las expresiones lambda de C++11 también pueden usarse en donde se utilizan apuntadores a funciones y objetos de función.

Ventajas de los objetos de función sobre los apuntadores a funciones

Los *objetos de función* ofrecen varias ventajas en comparación con los *apuntadores a funciones*. El compilador puede poner en línea el operador `operator()` sobrecargado de un *objeto de función* para mejorar el rendimiento. Además, como son objetos de clases, los *objetos de función* pueden tener miembros de datos que `operator()` puede utilizar para realizar su tarea.

Objetos función predefinidos de la Biblioteca de plantillas estándar

Muchos *objetos de función* predefinidos se pueden encontrar en el encabezado **`<functional>`**. En la figura 16.14 se enlistan varias de las docenas de *objetos de función* de la Biblioteca estándar, todos los cuales se implementan como plantillas de clases. La sección 20.8 del estándar de C++ contiene la lista completa de objetos de función. Utilizamos el *objeto de función* `less< T >` en los ejemplos con `set`, `multiset` y `priority_queue`, para especificar el orden de los elementos en un contenedor.

Uso del algoritmo **`accumulate`**

En la figura 16.15 se demuestra el uso del algoritmo numérico `accumulate` (presentado en la figura 16.15) para calcular la suma de los cuadrados de los elementos en un `array`. El cuarto argumento para `accumulate` es un **objeto de función binaria** (es decir, un *objeto de función* para el que `operator()` recibe dos argumentos) o un *apuntador* a una **función binaria** (es decir, una función que recibe dos argumentos). La función `accumulate` se demuestra dos veces: una vez con un *apuntador a una función* y la otra con un *objeto de función*.

Objeto de función	Tipo	Objeto de función	Tipo
`divides< T >`	aritmético	`logical_or< T >`	lógico
`equal_to< T >`	relacional	`minus< T >`	aritmético
`greater< T >`	relacional	`modulus< T >`	aritmético
`greater_equal< T >`	relacional	`negate< T >`	aritmético
`less< T >`	relacional	`not_equal_to< T >`	relacional
`less_equal< T >`	relacional	`plus< T >`	aritmético
`logical_and< T >`	lógico	`multiplies< T >`	aritmético
`logical_not< T >`	lógico		

Fig. 16.14 | Objetos de función de la Biblioteca estándar.

```
1  // Fig. 16.15: fig16_15.cpp
2  // Demostración de los objetos de función.
3  #include <iostream>
4  #include <array> // definición de la plantilla de clase array
5  #include <algorithm> // algoritmo de copia
6  #include <numeric> // algoritmo accumulate
7  #include <functional> // definición de binary_function
8  #include <iterator> // ostream_iterator
9  using namespace std;
10
11 // función binaria que suma el cuadrado de su segundo argumento y el
12 // total actual en su primer argumento, y después devuelve la suma
13 int sumarCuadrados( int total, int valor )
14 {
15    return total + valor * valor;
16 } // fin de la función sumarCuadrados
17
18 // La plantilla de clase ClaseSumarCuadrados define el operator() sobrecargado
19 // que suma el cuadrado de su segundo argumento y el total actual
20 // en su primer argumento, y después devuelve la suma
21 template< typename T >
22 class class ClaseSumarCuadrados
23 {
24 public:
25    // suma el cuadrado de valor al total y devuelve el resultado
26    T operator()( const T &total, const T &valor )
27    {
28       return total + valor * valor;
29    } // fin de la función operator()
30 }; // fin de la clase ClaseSumarCuadrados
31
32 int main()
33 {
34    const size_t TAMANIO = 10;
35    array< int, TAMANIO > enteros = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Fig. 16.15 | Objeto de función binaria (parte 1 de 2).

```
36     ostream_iterator< int > salida( cout, " " );
37
38     cout << "el arreglo enteros contiene:\n";
39     copy( enteros.cbegin(), enteros.cend(), salida );
40
41     // calcula la suma de los cuadrados de los elementos del arreglo
42     // enteros usando la función binaria sumarCuadrados
43     int resultado = accumulate( enteros.cbegin(), enteros.cend(),
44       0, sumarCuadrados );
45
46     cout << "\n\nSuma de los cuadrados de los elementos en enteros usando "
47       << "la funcion\nbinaria sumarCuadrados: "  << resultado;
48
49     // calcula la suma de los cuadrados de los elementos del arreglo
50     // enteros usando el objeto de función binaria
51     resultado = accumulate( enteros.cbegin(), enteros.cend(),
52       0, ClaseSumarCuadrados< int >() );
53
54     cout << "\n\nSuma de los cuadrados de los elementos en enteros usando "
55       << "el objeto\nde funcion binaria de tipo "
56       << "ClaseSumarCuadrados< int >: " << resultado << endl;
57  } // fin de main
```

```
el arreglo enteros contiene:
1 2 3 4 5 6 7 8 9 10

Suma de los cuadrados de los elementos en enteros usando la funcion
binaria sumarCuadrados: 385

Suma de los cuadrados de los elementos en enteros usando el objeto
de funcion binaria de tipo ClaseSumarCuadrados< int >: 385
```

Fig. 16.15 | Objeto de función binaria (parte 2 de 2).

Función `sumarCuadrados`

En las líneas 13 a 16 se define la función `sumarCuadrados`, que calcula el cuadrado de su segundo argumento llamado `valor`, suma ese cuadrado con su primer argumento `total` y devuelve la suma. La función `accumulate` pasará como el segundo argumento para `sumarCuadrados` a cada uno de los elementos de la secuencia sobre la cual va a iterar en el ejemplo. En la primera llamada a `sumarCuadrados`, el primer argumento será el valor inicial del `total` (que se suministra como el tercer argumento para `accumulate`; 0 en este programa). Todas las llamadas subsecuentes a `sumarCuadrados` reciben como primer argumento la suma actual devuelta por la llamada anterior a `sumarCuadrados`. Cuando `accumulate` termina, devuelve la suma de los cuadrados de todos los elementos en la secuencia.

Clase `ClaseSumarCuadrados`

En las líneas 21 a 30 se define la plantilla de clase `ClaseSumarCuadrados` con un `operator()` sobrecargado que tiene dos parámetros y devuelve un valor: los requerimientos para un objeto de función binario. En la primera llamada al *objeto de función*, el primer argumento será el valor inicial del `total` (que se suministra como el tercer argumento para `accumulate`; 0 en este programa) y el segundo argumento será el primer elemento en el `array enteros`. Todas las llamadas subsiguientes a `operator` reciben como primer argumento el resultado devuelto por la llamada anterior al *objeto de función*, y el segundo argu-

mento será el siguiente elemento en el `array`. Cuando `accumulate` termina de ejecutarse, devuelve la suma de los cuadrados de todos los elementos en el `array`.

***Pasar apuntadores a funciones y objetos de función al algoritmo* `accumulate`**
En las líneas 43 y 44 se hace una llamada a la función `accumulate` con un *apuntador a la función* `sumarCuadrados` como su último argumento. De manera similar, la instrucción en las líneas 51 y 52 llama a la función `accumulate` con un objeto de la clase `ClaseSumarCuadrados` como el último argumento. La expresión `ClaseSumarCuadrados<int>()` crea (y llama al constructor determinado para) una instancia de la clase `ClaseSumarCuadrados` (un *objeto de función*) que se pasa a `accumulate`, la cual invoca a la función `operator()`. Las líneas 51 y 52 podrían haberse escrito como dos instrucciones separadas, como se muestra a continuación:

```
ClaseSumarCuadrados< int > objetoSumarCuadradost;
resultado = accumulate( enteros.cbegin(), enteros.cend(),
  0, objetoSumarCuadrados );
```

En la primera línea se define un objeto de la clase `ClaseSumarCuadrados`. Ese objeto se pasa a continuación a la función `accumulate`.

16.5 Expresiones lambda

Como hemos visto en este capítulo, muchos algoritmos pueden recibir apuntadores a funciones u objetos de función como parámetros. Antes de poder pasar un apuntador a función o un objeto de función a un algoritmo, hay que declarar la función o clase correspondiente.

Las **expresiones Lambda** (o **funciones lambda**) de C++11 nos permiten definir objetos de función anónimos *en donde se pasan* a una función. Se definen en forma local dentro de las funciones y pueden "capturar" (por valor o por referencia) las variables locales de la función circundante, y luego manipulan estas variables en el cuerpo de la expresión lambda. En la figura 16.16 se demuestra un ejemplo simple de una expresión lambda que duplica el valor de cada elemento en un arreglo `int`.

```
1  // Fig. 16.16: fig16_16.cpp
2  // Expresiones lambda.
3  #include <iostream>
4  #include <array>
5  #include <algorithm>
6  using namespace std;
7
8  int main()
9  {
10   const size_t TAMANIO = 4; // tamaño del arreglo valores
11   array< int, TAMANIO >  valores = { 1, 2, 3, 4 }; // inicializa valores
12
13   // imprime cada elemento multiplicado por dos
14   for_each( valores.cbegin(), valores.cend(),
15     []( int i ) { cout << i * 2 << endl; }  );
16
17   int suma = 0; // inicializa suma con cero
18
19   // suma cada elemento a la variable suma
20   for_each( valores.cbegin(), valores.cend(),
21     [ &suma ]( int i ) { suma += i; } );
```

Fig. 16.16 | Expresiones lambda (parte 1 de 2).

```
22
23     cout << "La suma es " << suma << endl; // imprime suma
24   } // fin de main
```

```
2
4
6
8
La suma es 10
```

Fig. 16.16 | Expresiones lambda (parte 2 de 2).

En las líneas 10 y 11 se declara e inicializa un pequeño contenedor `array` de valores `int` llamado `valores`. En las líneas 14 y 15 se invoca el algoritmo `for_each` en los elementos de `valores`. El tercer argumento (línea 15) para `for_each` es una *expresión lambda*. Las expresiones lambda comienzan con un *introductor lambda* (`[]`) seguido de una lista de parámetros y el cuerpo de la función. Los tipos de valores de retorno pueden inferirse de manera automática si el cuerpo es una sola instrucción de la forma `return` *expresión*. De lo contrario, el tipo de valor de retorno es `void` de manera predeterminada, o podemos usar explícitamente un *tipo de valor de retorno al final* (introducido en la sección 6.19). El compilador convierte la expresión lambda en un objeto de función. La expresión lambda en la línea 15 recibe un `int`, lo multiplica por 2 y muestra el resultado. El algoritmo `for_each` pasa cada elemento del `array` a la expresión lambda.

La segunda llamada al algoritmo `for_each` (líneas 20 y 21) calcula la suma de los elementos del `array`. El introductor lambda `[&suma]` indica que esta expresión lambda *captura* la variable local `suma` *por referencia* (observe el uso del símbolo `&`), de modo que la expresión lambda pueda modificar el valor de `suma`. Sin el símbolo `&`, `suma` se capturaría por valor y *no* se actualizaría la variable local fuera de la expresión lambda. El algoritmo `for_each` pasa cada elemento de `valores` a la expresión lambda, que suma el valor a la variable `suma`. Después, en la línea 23 se muestra el valor de `suma`.

Es posible asignar expresiones lambda a variables, que luego pueden usarse para invocar la expresión lambda o pasarla a otras funciones. Por ejemplo, podemos asignar la expresión lambda en la línea 15 a una variable como se muestra a continuación:

```
auto miLambda = []( int i ) { cout << i * 2 << endl; };
```

Después podemos usar el nombre de la variable como el nombre de una función para invocar a la expresión lambda, como en el siguiente ejemplo:

```
miLambda( 10 ); // imprime 20
```

16.6 Resumen de algoritmos de la Biblioteca estándar

El estándar de C++ especifica más de 90 algoritmos; muchos de ellos están sobrecargados con dos o más versiones. El estándar separa los algoritmos en varias categorías: *algoritmos de secuencia mutante*, *algoritmos de secuencia no modificadores*, *algoritmos de ordenamiento y relacionados*, y *operaciones numéricas generalizadas*. Para aprender sobre los algoritmos que no presentamos en este capítulo, consulte la documentación de su compilador o visite sitios como

```
en.cppreference.com/w/cpp/algorithm
msdn.microsoft.com/en-us/library/yah1y2x8.aspx
```

Algoritmos de secuencia mutantes

En la figura 16.17 se muestran muchos de los **algoritmos de secuencia mutante**; es decir, algoritmos que modifican los contenedores en los que operan. Los algoritmos nuevos en C++11 están marcados con un * en las figuras 16.17 a 16.20. Los algoritmos que se presentan en este capítulo aparecen en **negrita**.

Algoritmos de secuencia mutante del encabezado <algorithm>			
copy	**copy_n***	**copy_if***	**copy_backward**
move*	move_backward*	**swap**	**swap_ranges**
iter_swap	**transform**	**replace**	**replace_if**
replace_copy	**replace_copy_if**	**fill**	**fill_n**
generate	**generate_n**	**remove**	**remove_if**
remove_copy	**remove_copy_if**	**unique**	**unique_copy**
reverse	**reverse_copy**	rotate	rotate_copy
random_shuffle	shuffle*	is_partitioned*	partition
stable_partition	partition_copy*	partition_point*	

Fig. 16.17 | Algoritmos de secuencia mutante del encabezado <algorithm>.

Algoritmos de secuencia no modificadores

La figura 16.18 muestra los **algoritmos de secuencia no modificadores**; es decir, algoritmos que *no* modifican los contenedores en los que operan.

Algoritmos de secuencia no modificadores del encabezado <algorithm>			
all_of*	**any_of***	**none_of***	**for_each**
find	**find_if**	**find_if_not***	find_end
find_first_of	adjacent_find	**count**	**count_if**
mismatch	**equal**	is_permutation*	search
search_n			

Fig. 16.18 | Algoritmos de secuencia no modificadores del encabezado <algorithm>.

Algoritmos de ordenamiento y relacionados

La figura 16.19 muestra los *algoritmos de ordenamiento y relacionados*.

Algoritmos de ordenamiento y relacionados del encabezado <algorithm>			
sort	stable_sort	partial_sort	partial_sort_copy
is_sorted*	is_sorted_until*	nth_element	**lower_bound**
upper_bound	**equal_range**	**binary_search**	**merge**

Fig. 16.19 | Algoritmos de ordenamiento y relacionados del encabezado <algorithm> (parte 1 de 2).

Algoritmos de ordenamiento y relacionados del encabezado <algorithm>			
inplace_merge	includes	set_union	set_intersection
set_difference	set_symmetric_difference		push_heap
pop_heap	make_heap	sort_heap	is_heap*
is_heap_until*	min	max	minmax*
min_element	max_element	minmax_element*	lexicographical_compare
next_permutation	prev_permutation		

Fig. 16.19 | Algoritmos de ordenamiento y relacionados del encabezado <algorithm> (parte 2 de 2).

Algoritmos numéricos

La figura 16.20 muestra los algoritmos numéricos del encabezado **<numeric>**.

Algoritmos numéricos del encabezado <numeric>		
accumulate	partial_sum	iota*
inner_product	adjacent_difference	

Fig. 16.20 | Algoritmos numéricos del encabezado <algorithm>.

16.7 Conclusión

En este capítulo demostramos muchos de los algoritmos de la Biblioteca estándar, incluyendo los algoritmos matemáticos, los algoritmos básicos de búsqueda y ordenamiento, y las operaciones establecer (set). Aprendió sobre los tipos de iteradores que requiere cada algoritmo, y que cada algoritmo puede usarse con cualquier contenedor que soporte la mínima funcionalidad de iteradores que requiera el algoritmo. Presentamos los objetos de función que trabajan en forma sintáctica y semántica como las funciones ordinarias, pero ofrecen ventajas tales como el rendimiento y la habilidad de almacenar datos. Por último, utilizamos expresiones lambda para crear objetos de funciones en línea y luego los pasamos a los algoritmos de la Biblioteca estándar.

Anteriormente en el libro presentamos el manejo de excepciones, en nuestra discusión sobre los arreglos. En el siguiente capítulo veremos con más detalle el extenso conjunto de herramientas para manejar excepciones de C++.

Resumen

Sección 16.1 Introducción

- Los algoritmos de la Biblioteca estándar son funciones que realizan manipulaciones de datos comunes, como búsqueda, ordenamiento y comparación de elementos o contenedores completos.

Sección 16.3.1 `fill`, `fill_n`, `generate` y `generate_n`

- Los algoritmos `fill` y `fill_n` (pág. 693) establecen cada elemento en un rango de elementos del contenedor en un valor específico.

- Los algoritmos `generate` y `generate_n` (pág. 693) utilizan una función generadora (pág. 693) o un objeto de función para crear valores para cada elemento en un rango de elementos del contenedor.

Sección 16.3.2 `equal`, `mismatch` y `lexicographical_compare`

- El algoritmo `equal` (pág. 696) compara la igualdad entre dos secuencias de valores.
- El algoritmo `mismatch` (pág. 697) compara dos secuencias de valores y devuelve un par de iteradores que indican la ubicación en cada secuencia de los elementos que no coinciden.
- El algoritmo `lexicographical_compare` (pág. 697) compara el contenido de dos secuencias.

Sección 16.3.3 `remove`, `remove_if`, `remove_copy` y `remove_copy_if`

- El algoritmo `remove` (pág. 699) elimina todos los elementos con un valor específico en cierto rango.
- El algoritmo `remove_copy` (pág. 699) copia todos los elementos que no tienen un valor específico en cierto rango.
- El algoritmo `remove_if` (pág. 699) elimina todos los elementos que cumplen con la condición `if` en cierto rango.
- El algoritmo `remove_copy_if` (pág. 700) copia todos los elementos que cumplen con la condición `if` en cierto rango.

Sección 16.3.4 `replace`, `replace_if`, `replace_copy` y `replace_copy_if`

- El algoritmo `replace` (pág. 702) reemplaza todos los elementos con un valor específico en cierto rango.
- El algoritmo `replace_copy` (pág. 702) copia todos los elementos en un rango y reemplaza todos los elementos de cierto valor con un valor distinto.
- El algoritmo `replace_if` (pág. 702) reemplaza todos los elementos que cumplen con la condición `if` en cierto rango.
- El algoritmo `replace_copy_if` (pág. 702) copia todos los elementos en un rango y reemplaza todos los elementos que cumplen con la condición `if` en cierto rango.

Sección 16.3.5 Algoritmos matemáticos

- El algoritmo `random_shuffle` (pág. 704) reordena al azar los elementos en cierto rango.
- El algoritmo `count` (pág. 705) cuenta los elementos con un valor específico en cierto rango.
- El algoritmo `count_if` (pág. 705) cuenta los elementos que cumplen con la condición `if` en cierto rango.
- El algoritmo `min_element` (pág. 705) localiza el elemento más pequeño en cierto rango.
- El algoritmo `max_element` (pág. 705) localiza el elemento más grande en cierto rango.
- El algoritmo `minmax_element` (pág. 705) localiza los elementos mayor y menor en cierto rango.
- El algoritmo `accumulate` (pág. 705) suma los valores en cierto rango.
- El algoritmo `for_each` (pág. 706) aplica una función general o un objeto de función a cada elemento en un rango.
- El algoritmo `transform` (pág. 706) aplica una función general o un objeto de función a cada elemento en un rango, y reemplaza cada elemento con el resultado de la función.

Sección 16.3.6 Algoritmos básicos de búsqueda y ordenamiento

- El algoritmo `find` (pág. 708) localiza un valor específico en cierto rango.
- El algoritmo `find_if` (pág. 709) localiza el primer valor en cierto rango que cumpla con la condición `if`.
- El algoritmo `sort` (pág. 709) ordena los elementos en cierto rango en orden ascendente, o en un orden especificado por un predicado.
- El algoritmo `binary_search` (pág. 709) determina si un valor específico está en un rango ordenado de elementos.

- El algoritmo `all_of` (pág. 709) determina si una función predicado unaria devuelve `true` para todos los elementos en el rango.
- El algoritmo `any_of` (pág. 709) determina si una función predicado unaria devuelve `true` para alguno de los elementos en el rango.
- El algoritmo `none_of` (pág. 709) determina si una función predicado unaria devuelve `false` para todos los elementos en el rango.
- El algoritmo `find_if_not` (pág. 710) localiza el primer valor en cierto rango que no cumpla con la condición de la instrucción `if`.

Sección 16.3.7 `swap`, `iter_swap` y `swap_ranges`

- El algoritmo `swap` (pág. 711) intercambia dos valores.
- El algoritmo `iter_swap` (pág. 711) intercambia los dos elementos a los que apuntan los dos argumentos iteradores.
- El algoritmo `swap_ranges` (pág. 711) intercambia los elementos en cierto rango.

Sección 16.3.8 `copy_backward`, `merge`, `unique` y `reverse`

- El algoritmo `copy_backward` (pág. 712) copia los elementos en un rango y los coloca en un contenedor, empezando desde el final y avanzando hacia la parte inicial.
- El algoritmo `move` (pág. 713) mueve los elementos en un rango de un contenedor a otro.
- El algoritmo `move_backward` (pág. 713) mueve los elementos en un rango de un contenedor a otro, empezando desde el final y avanzando hacia la parte inicial.
- El algoritmo `merge` (pág. 713) combina dos secuencias en orden ascendente de valores en una tercera secuencia en orden ascendente.
- El algoritmo `unique` (pág. 713) elimina los elementos duplicados en cierto rango de una secuencia ordenada.
- El algoritmo `copy_if` (pág. 714) copia cada elemento de un rango, si una función predicado unaria devuelve `true` para ese elemento.
- El algoritmo `reverse` (pág. 714) invierte todos los elementos en cierto rango.
- El algoritmo `copy_n` (pág. 714) copia un número especificado de elementos, empezando desde una ubicación especificada y los coloca en un contenedor, empezando en la ubicación especificada.

Sección 16.3.9 `inplace_merge`, `unique_copy` y `reverse_copy`

- El algoritmo `inplace_merge` (pág. 715) combina dos secuencias ordenadas de elementos en el mismo contenedor.
- El algoritmo `unique_copy` (pág. 715) realiza una copia de todos los elementos únicos en la secuencia ordenada de valores en cierto rango.
- El algoritmo `reverse_copy` (pág. 715) realiza una copia inversa de todos los elementos en cierto rango.

Sección 16.3.10 Operaciones establecer (set)

- El algoritmo `includes` de `set` (pág. 717) compara dos conjuntos de valores ordenados para determinar si cada elemento del segundo objeto `set` está en el primer objeto `set`.
- El algoritmo `set_difference` de `set` (pág. 718) busca los elementos del primer conjunto `set` de valores ordenados que no estén en el segundo conjunto `set` de valores ordenados (ambos conjuntos de valores deben estar en orden ascendente).
- El algoritmo `set_intersection` de `set` (pág. 718) determina los elementos del primer conjunto `set` de valores ordenados que estén en el segundo conjunto `set` de valores ordenados (ambos conjuntos de valores deben estar en orden ascendente).
- El algoritmo `set_symmetric_difference` de `set` (pág. 718) determina los elementos en el primer conjunto `set` que no estén en el segundo conjunto `set`, y los elementos en el segundo conjunto `set` que no estén en el primer conjunto `set` (ambos conjuntos de valores deben estar en orden ascendente).

- El algoritmo `set_union` de `set` (pág. 719) crea un conjunto `set` de todos los elementos que estén en uno o ambos conjuntos `set` ordenados (ambos conjuntos de valores deben estar en orden ascendente).

Sección 16.3.11 `lower_bound`, `upper_bound` y `equal_range`

- El algoritmo `lower_bound` (pág. 721) busca la primera ubicación en una secuencia ordenada de valores en la que el tercer argumento se debe insertar en la secuencia, de tal forma que la secuencia siga almacenada en orden ascendente.
- El algoritmo `upper_bound` (pág. 721) busca la última ubicación en una secuencia ordenada de valores en los que el tercer argumento se podría insertar en la secuencia, de tal forma que ésta siga almacenada en orden ascendente.
- El algoritmo `equal_range` (pág. 721) devuelve el límite inferior y el límite superior como un objeto `pair`.

Sección 16.3.12 Ordenamiento de montón (heapsort)

- El algoritmo `make_heap` (pág. 723) recibe una secuencia de valores en un cierto rango, y crea un montón que se puede utilizar para producir una secuencia ordenada.
- El algoritmo `sort_heap` (pág. 723) ordena una secuencia de valores en cierto rango de un montón.
- El algoritmo `pop_heap` (pág. 724) elimina el elemento superior del montón.
- El algoritmo `is_heap` (pág. 724) devuelve `true` si los elementos en el rango especificado representan un montón.
- El algoritmo `is_heap_until` (pág. 724) comprueba el rango especificado de valores y devuelve un iterador que apunta al último elemento en el rango para el que los elementos hasta pero *sin* incluir) ese iterador representan un montón.

Sección 16.3.13 `min`, `max`, `minmax` y `minmax_element`

- Los algoritmos `min` y `max` (pág. 275) determinan el mínimo de dos elementos y el máximo de dos elementos, respectivamente.
- Ahora C++11 incluye versiones sobrecargadas de los algoritmos `min` y `max`, cada una de las cuales recibe un parámetro `initializer_list` y devuelve el elemento más pequeño o más grande en el inicializador de lista que se pasa como argumento. Cada algoritmo está sobrecargado con una versión que recibe como segundo argumento una *función predicado binaria* para comparar valores.
- Ahora C++11 incluye el algoritmo `minmax` (pág. 725), que recibe dos elementos y devuelve un `pair` en donde el elemento más pequeño se almacena en `first` y el elemento más grande se almacena en `second`. Una segunda versión de este algoritmo recibe como tercer argumento una función predicado binaria para comparar valores.
- Ahora C++11 incluye el algoritmo `minmax_element` (pág. 705), el cual recibe dos iteradores de entrada que representan un rango de elementos y devuelve un `pair` de iteradores en donde `first` apunta al elemento más pequeño en el rango y `second` apunta al más grande. Una segunda versión de este algoritmo recibe como tercer argumento una función predicado binaria para comparar valores.

Sección 16.4 Objetos de función

- Un objeto de función (pág. 726) es una instancia de una clase que sobrecarga a `operator()`.
- La Biblioteca estándar proporciona muchos objetos de función predefinidos, los cuales se encuentran en el encabezado `<functional>` (pág. 726).
- Los objetos de función binarios (pág. 726) reciben dos argumentos y devuelven un valor.

Sección 16.5 Expresiones lambda

- Las expresiones lambda (o funciones lambda; pág. 729) cuentan con una sintaxis simplificada para definir objetos de función de manera directa en donde se van a usar.

- Una función lambda puede capturar variables locales (por valor o por referencia) y manipularlas dentro del cuerpo de la función lambda.
- Las expresiones lambda comienzan con el introductor lambda `[]`, seguido de un parámetro y el cuerpo de la función. Los tipos de valores de retorno pueden inferirse de manera automática si el cuerpo es una sola instrucción de la forma `return` *expresión*. De lo contrario, el tipo de valor de retorno es `void` de manera predeterminada.
- Para capturar una variable local, hay que especificarla en el introductor lambda. Para capturar por referencia, use el símbolo `&`.

Ejercicios de autoevaluación

16.1 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Los algoritmos de la Biblioteca estándar pueden operar en arreglos tipo C basados en apuntador.
b) Los algoritmos de la Biblioteca estándar están encapsulados como funciones miembro dentro de cada clase de contenedor.
c) Al usar el algoritmo `remove` en un contenedor, el algoritmo no reduce el tamaño del contenedor del que se van a eliminar elementos.
d) Una desventaja de usar los algoritmos de la Biblioteca estándar es que dependen de los detalles de implementación de los contenedores en los que operan
e) El algoritmo `remove_if` no modifica el número de elementos en el contenedor, sino que mueve al principio del contenedor todos los elementos que no se eliminen.
f) El algoritmo `find_if_not` localiza todos los valores en el rango para el que la función predicado unaria especificada devuelva `false`.
g) Use el algoritmo `set_union` para crear un conjunto de todos los elementos que estén en uno o en los dos conjuntos ordenados (ambos conjuntos de valores deben estar en orden ascendente).

16.2 Llene los espacios en blanco en cada uno de los siguientes enunciados.

a) Los algoritmos de la Biblioteca estándar operan de manera indirecta sobre los elementos de los contenedores, usando _____.
b) El algoritmo `sort` requiere un iterador _____.
c) Los algoritmos _____ y _____ establecen cada elemento en un rango de elementos de contenedores a un valor específico.
d) El algoritmo _____ compara la igualdad de dos secuencias de valores.
e) El algoritmo _____ de C++11 localiza los elementos menor y mayor en un rango.
f) Un `back_inserter` llama a la función _____ predeterminada del contenedor para insertar un elemento en la parte final del mismo. Si se inserta un elemento en un contenedor que no tenga más espacio disponible, el contenedor aumenta su tamaño.
g) Cualquier algoritmo que pueda recibir un apuntador a función también puede recibir un objeto de una clase que sobrecargue al operador paréntesis con una función llamada `operator()`, siempre y cuando el operador sobrecargado cumpla los requerimientos del algoritmo. Un objeto de dicha clase se conoce como _____ y puede usarse en forma sintáctica y semántica como una función o apuntador a función.

16.3 Escriba una instrucción para realizar cada una de las siguientes tareas:

a) Use el algoritmo `fill` para llenar todo el `array` de objetos `string` llamado `elementos` con `"hola"`.
b) La función `siguienteInt` devuelve el siguiente valor `int` en la secuencia, empezando con `0` la primera vez que se llama. Use el algoritmo `generate` y la función `siguienteInt` para llenar el contenedor `array` de valores `int` llamado `enteros`.
c) Use el algoritmo `equal` para comparar la igualdad entre dos listas (`cadenas1` y `cadenas2`). Almacene el resultado en la variable `bool resultado`
d) Use el algoritmo `remove_if` para eliminar del `vector` de objetos `string` llamado `colores` todas las cadenas que comiencen con `"bl"`. La función `comienzaConBL` devuelve `true` si su argumento `string` comienza con `"bl"`. Almacene el iterador que devuelva el algoritmo en `nuevoUltimoElemento`.

e) Use el algoritmo `replace_if` para reemplazar con `0` todos los elementos con valores mayores a `100` en el `array` de valores `int` llamado `valores`. La función `mayorQue100` devuelve `true` si su argumento es mayor que `100`.
f) Use el algoritmo `minmax_element` para encontrar los valores menor y mayor en el `array` de valores `double` llamado `temperaturas`. Almacene el `pair` de iteradores que se devuelva en `resultado`.
g) Use el algoritmo `sort` para ordenar el `array` de objetos `string` llamado `colores`.
h) Use el algoritmo `reverse` para invertir el orden de los elementos en el `array` de objetos `string` llamado `colores`.
i) Use el algoritmo `merge` para combinar el contenido de los dos contenedores `array` ordenados, llamados `valores1` y `valores2`, en un tercer `array` llamado `resultados`.
j) Escriba una expresión lambda que devuelva el cuadrado de su argumento `int` y asigne la expresión lambda a la variable `cuadradoInt`.

Respuestas a los ejercicios de autoevaluación

16.1 a) Verdadero.
b) Falso. Los algoritmos de la STL no son funciones miembro. Operan de manera indirecta en los contenedores a través de los iteradores.
c) Verdadero.
d) Falso. Los algoritmos de la Biblioteca estándar no dependen de los detalles de implementación de los contenedores en los que operan.
e) Verdadero.
f) Falso. Sólo localiza el primer valor en el rango para el que la función predicado unaria especificada devuelva `false`.
g) Verdadero.

16.2 a) Iteradores. b) de acceso aleatorio. c) `fill`, `fill_n`. d) `equal`. e) `minmax_element`. f) `push_back`. g) objeto de función.

16.3 a)
```
fill( elementos.begin(), elementos.end(), "hola" );
```
b)
```
generate( enteros.begin(), enteros.end(), siguienteInt );
```
c)
```
bool resultado =
   equal( cadenas1.cbegin(), cadenas1.cend(), cadenas2.cbegin() );
```
d)
```
auto nuevoUltimoElemento =
   remove_if( colores.begin(), colores.end(), comienzaConBL );
```
e)
```
replace_if( valores.begin(), valores.end(), mayorQue100 );
```
f)
```
auto resultado =
   minmax_element( temperaturas.cbegin(), temperaturas.cend() );
```
g)
```
sort( colores.begin(), colores.end() );
```
h)
```
reverse( colores.begin(), colores.end() );
```
i)
```
merge( valores1.cbegin(), valores1.cend(), valores2.cbegin(), valores2.cend(),
   resultados.begin() );
```
j)
```
auto cuadradoInt = []( int i ) { return i * i; };
```

Ejercicios

16.4 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Como los algoritmos de la Biblioteca estándar procesan contenedores de manera directa, es común que un algoritmo pueda usarse con muchos contenedores diferentes.
b) Use el algoritmo `for_each` para aplicar una función general a cada elemento en un rango; `for_each` no modifica la secuencia.

c) De manera predeterminada, el algoritmo `sort` ordena los elementos en un rango, en orden ascendente.
d) Use el algoritmo `merge` para formar una nueva secuencia, colocando la segunda secuencia después de la primera.
e) Use el algoritmo `set_intersection` para buscar los elementos de un primer conjunto de valores ordenados que no estén en un segundo conjunto de valores ordenados (ambos conjuntos de valores deben estar en orden ascendente).
f) Los algoritmos `lower_bound`, `upper_bound` y `equal_range` se utilizan comúnmente para localizar puntos de inserción en secuencias ordenadas.
g) Las expresiones lambda también pueden usarse en donde se utilizan apuntadores a funciones y objetos de función en los algoritmos.
h) Las expresiones Lambda de C++11 se definen de manera local dentro de las funciones y pueden "capturar" (por valor o por referencia) las variables locales de la función circundante, para después manipular estas variables en el cuerpo de la expresión lambda.

16.5 Complete los siguientes enunciados.
a) Mientras que el (los) _____ de un contenedor (o arreglo integrado) cumpla(n) con los requerimientos de un algoritmo, éste podrá trabajar en el contenedor.
b) Los algoritmos `generate` y `generate_n` usan una función _____ para crear valores para cada elemento en un *rango* de elementos de un contenedor. Ese tipo de función no recibe argumentos y devuelve un valor que puede colocarse en un elemento del contenedor.
c) Los apuntadores a arreglos integrados son iteradores _____.
d) Use el algoritmo _____ (cuya plantilla se encuentra en el encabezado `<numeric>`) para sumar los valores en un rango.
e) Use el algoritmo _____ para aplicar una función general a cada elemento en un rango cuando necesite modificar esos elementos.
f) Para trabajar correctamente, el algoritmo `binary_search` requiere que la secuencia de valores sea _____.
g) Use la función `iter_swap` para intercambiar los elementos a los que apunten dos iteradores _____ y para intercambiar los valores en esos elementos.
h) Ahora C++11 incluye el algoritmo `minmax` que recibe dos elementos y devuelve un(a) _____ en donde el elemento más pequeño se almacena en `first` y el elemento más grande se almacena en `second`.
i) Los algoritmos _____ modifican los contenedores en los que operan.

16.6 Liste varias ventajas que ofrecen los objetos de función sobre los apuntadores a funciones.

16.7 ¿Qué ocurre si aplicamos el algoritmo `unique` a una secuencia ordenada de elementos en un rango?

16.8 ***(Eliminación de duplicados)*** Lea 20 enteros y colóquelos en un contenedor `array`. A continuación, use el algoritmo `unique` para reducir el `array` a los valores únicos introducidos por el usuario. Use el algoritmo `copy` para mostrar los valores únicos.

16.9 ***(Eliminación de duplicados)*** Modifique el ejercicio 16.8 para usar el algoritmo `unique_copy`. Los valores únicos deben insertarse en un `vector` que en un principio este vacío. Use un `back_inserter` para permitir que el `vector` aumente su tamaño a medida que se agreguen nuevos elementos. Use el algoritmo `copy` para mostrar los valores únicos.

16.10 ***(Leer datos de un archivo)*** Use un `istream_iterator<int>`, el algoritmo `copy` y un `back_inserter` para leer el contenido de un archivo de texto que contenga valores `int` separados por espacio en blanco. Coloque los valores `int` en un `vector` de valores `int`. El primer argumento para el algoritmo `copy` debe ser el objeto

`istream_iterator<int>` asociado con el objeto `ifstream` del archivo de texto. El segundo argumento debe ser un objeto `istream_iterator<int>` que se inicialice mediante el uso del constructor predeterminado de la plantilla de clase `istream_iterator`; el objeto resultante puede usarse como iterador "final". Después de leer el contenido del archivo, muestre el contenido del `vector` resultante.

16.11 ***(Combinar listas ordenadas)*** Escriba un programa que utilice los algoritmos de la Biblioteca estándar para combinar dos contenedores `list` ordenados de objetos `string` en un solo contenedor `list` ordenado de objetos `string`, y que luego muestre el contenedor `list` resultante.

16.12 ***(Probador de palíndromos)*** Un palíndromo es una cadena (`string`) que se escribe de la misma forma al derecho y al revés. Algunos ejemplos de palíndromos son "radar" y "Anita lava la tina". Escriba una función llamada `probadorPalindromos` que utilice el algoritmo `reverse` en una copia de un objeto `string` y luego compare el objeto `string` original con el `string` invertido para determinar si el original es un palíndromo o no. Al igual que los contenedores de la Biblioteca estándar, los objetos `string` proporcionan funciones como `begin` y `end` para obtener iteradores que apuntan a caracteres en un objeto `string`. Suponga que el objeto `string` original contiene todas las letras en minúscula y no contiene signos de puntuación. Use la función `probadorPalindromos` en un programa.

16.13 ***(Probador de palíndromos mejorado)*** Mejore la función `probadorPalindromos` del ejercicio 16.12 para permitir cadenas que contengan letras mayúsculas y minúsculas, además de signos de puntuación. Antes de evaluar si la cadena original es un palíndromo, la función `probadorPalindromos` debe convertir el `string` a letras minúsculas y eliminar los signos de puntuación. Para simplificar, suponga que los únicos signos de puntuación pueden ser:

```
. , ! ; : ( )
```

Puede usar el algoritmo `copy_if y back_inserter` para crear una copia del objeto `string` original, eliminar los signos de puntuación y colocar los caracteres en un nuevo objeto `string`.

17

Manejo de excepciones: un análisis más detallado

Es cuestión de sentido común tomar un método y probarlo. Si falla, admítalo francamente y pruebe otro. Pero sobre todo, inténtelo.

—Franklin Delano Roosevelt

Si están corriendo y no saben hacia dónde se dirigen tengo que salir de alguna parte y atraparlos.

—Jerome David Salinger

Objetivos

En este capítulo aprenderá a:

- Usar `try`, `catch` y `throw` para detectar, manejar e indicar excepciones, respectivamente.
- Declarar nuevas clases de excepciones.
- Comprender cómo la limpieza de la pila permite que las excepciones que no se atrapan en un alcance, se atrapen en otro alcance.
- Manejar las fallas de `new`.
- Usar `unique_ptr` para prevenir las fugas de memoria.
- Comprender la jerarquía de excepciones estándar.

Plan general

17.1 Introducción

Como vimos en la sección 7.10, una **excepción** es la indicación de un problema que ocurre durante la ejecución de un programa. El **manejo de excepciones** nos permite crear aplicaciones que pueden resolver (o manejar) excepciones. En muchos casos, esto permite que un programa continúe su ejecución como si no se hubiera encontrado un problema. Las características que presentamos en este capítulo permiten a los programadores escribir **programas tolerantes a fallas** y **robustos**, que puedan tratar con los problemas sin dejar de ejecutarse, o que terminen de manera no dañina.

Comenzaremos con una revisión de los conceptos sobre manejo de excepciones mediante un ejemplo que demuestra cómo manejar una excepción que ocurre cuando una función intenta una división entre cero. Le mostraremos cómo manejar excepciones que ocurren en un constructor o destructor, y excepciones que ocurren si el operador `new` falla al asignar memoria para un objeto. Presentaremos varias clases de manejo de excepciones de la Biblioteca estándar de C++ y le mostraremos cómo crear sus propias clases.

Observación de Ingeniería de Software 17.1

El manejo de excepciones proporciona un mecanismo estándar *para procesar los errores. Esto es especialmente importante cuando se trabaja en un proyecto con un equipo extenso de programadores.*

Observación de Ingeniería de Software 17.2

Incorpore su estrategia de manejo de excepciones a su sistema desde su comienzo. Incluir un manejo efectivo de excepciones después de haber implementado un sistema puede ser difícil.

Tip para prevenir errores 17.1

Sin el manejo de excepciones, es común para una función calcular y devolver un valor al tener éxito, o devolver un indicador de error al fallar. Un problema común con esta arquitectura es usar el valor de retorno en un cálculo subsiguiente sin primero comprobar que el valor sea el indicador de error. El manejo de excepciones elimina este problema.

17.2 Ejemplo: manejo de un intento de dividir entre cero

Vamos a considerar un ejemplo simple de manejo de excepciones (figuras 17.1 y 17.2). Le mostraremos cómo manejar un problema aritmético común: la *división entre cero*. La división entre cero mediante el uso de aritmética de enteros por lo general hace que un programa termine en forma prematura. En la aritmética de punto flotante, muchas implementaciones de C++ permiten la división entre cero, en cuyo caso el resultado de infinito positivo o negativo se muestra como `INF` o `-INF`, respectivamente.

En este ejemplo definimos una función llamada `cociente`, la cual recibe dos enteros introducidos por el usuario, y divide su primer parámetro `int` entre su segundo parámetro `int`. Antes de realizar la división, la función convierte el valor del primer parámetro entero al tipo `double`. Después, el valor del segundo parámetro `int` se promueve (implícitamente) al tipo `double` para el cálculo. Así, la función `cociente` en realidad realiza la división utilizando dos valores `double` y devuelve un resultado `double`.

Aunque la división entre cero está permitida comúnmente en la aritmética de punto flotante, para el propósito de este ejemplo trataremos un intento de división entre cero como un error. Así, la función `cociente` evalúa su segundo parámetro para asegurar que no sea cero antes de permitir que continúe la división. Si el segundo parámetro es cero, la función *lanza una excepción* para indicar a la función que hizo la llamada que ocurrió un problema. Así, la función que hizo la llamada (`main` en este ejemplo) puede procesar la excepción y permitir que el usuario escriba dos nuevos valores, antes de llamar a la función `cociente` de nuevo. De esta forma, el programa puede seguir ejecutándose aún después de que se introduzca un valor inapropiado, con lo cual el programa se vuelve más robusto.

El ejemplo consiste en dos archivos. `ExcepcionDivisionEntreCero.h` (figura 17.1) define una *clase de excepción* que representa el tipo del problema que podría ocurrir en el ejemplo, y `fig17_02.cpp` (figura 17.2) define la función `cociente` y la función `main` que llama a la clase de excepción. La función `main` contiene el código que demuestra el manejo de excepciones.

Definición de una clase de excepción para representar el tipo del problema que podría ocurrir

En la figura 17.1 se define la clase `ExcepcionDivisionEntreCero` como una clase derivada de la clase **`runtime_error`** de la Biblioteca estándar (definida en el archivo de encabezado **`<stdexcept>`**). La clase `runtime_error`, una clase derivada de **`exception`** del encabezado **`<exception>`**), es la clase base estándar de C++ que representa los errores en tiempo de ejecución. La clase `exception` es la clase base estándar de C++ para la excepción en la Biblioteca estándar de C++. (En la sección 17.10 hablaremos sobre la clase `exception` y sus clases derivadas con detalle). Una clase de excepción típica que se deriva de la clase `runtime_error` define sólo un constructor (líneas 11 y 12) que pasa una cadena de mensaje de error al constructor de la clase base `runtime_error`. Cada clase de excepción que se deriva en forma directa o indirecta de `exception` contiene la función `virtual` **`what`**, la cual devuelve el mensaje de error de un objeto excepción. No es obligatorio derivar una clase de excepción personalizada (como `ExcepcionDivisionEntreCero`) de las clases de excepciones estándar que proporciona C++. Sin embargo, esto permite a los programadores utilizar la función `virtual what` para obtener un mensaje de error apropiado. En la figura 17.2 utilizamos un objeto de esta clase `ExcepcionDivisionEntreCero` para indicar cuándo se hace un intento de dividir entre cero.

```
1  // Fig. 17.1: ExcepcionDivisionEntreCero.h
2  // Definición de la clase ExcepcionDivisionEntreCero.
3  #include <stdexcept> // el encabezado stdexcept contiene runtime_error
4
5  // los objetos ExcepcionDivisionEntreCero deben lanzarse por las funciones
6  // al detectar las excepciones de división entre cero
7  class ExcepcionDivisionEntreCero : public std::runtime_error
8  {
9  public:
10    // el constructor especifica el mensaje de error predeterminado
11    ExcepcionDivisionEntreCero()
12       : std::runtime_error( "intento de dividir entre cero" ) {}
13 }; // fin de la clase ExcepcionDivisionEntreCero
```

Fig. 17.1 | Definición de la clase `ExcepcionDivisionEntreCero`.

Demostración del manejo de excepciones

El programa de la figura 17.2 utiliza el manejo de excepciones para envolver código que podría lanzar una `ExcepcionDivisionEntreCero` y para manejar esa excepción, en caso de que ocurra una. El usuario introduce dos enteros, que se pasan como argumentos a la función `cociente` (líneas 10 a 18). Esta función divide su primer parámetro (`numerador`) entre su segundo parámetro (`denominador`). Suponiendo que el usuario no especifica `0` como el denominador para la división, la función `cociente` devuelve el resultado de la división. Si el usuario introduce `0` para el denominador, `cociente` lanza una excepción. En los resultados de ejemplo, las primeras dos líneas muestran un cálculo exitoso y las siguientes dos líneas muestran un cálculo fallido, debido a un intento de dividir entre cero. Cuando ocurre la excepción, el programa informa al usuario del error y le pide que introduzca dos nuevos enteros. Una vez que hablemos sobre el código, consideraremos las entradas del usuario y el flujo de control del programa que producen *estos resultados*.

```
1  // Fig. 17.2: fig17_02.cpp
2  // Un ejemplo que lanza excepciones
3  // intentar dividir entre cero.
4  #include <iostream>
5  #include "ExcepcionDivisionEntreCero.h" // clase ExcepcionDivisionEntreCero
6  using namespace std;
7
8  // realiza la división y lanza un objeto ExcepcionDivisionEntreCero si
9  // ocurre una excepción de división entre cero
10 double cociente( int numerador, int denominador )
11 {
12    // lanza ExcepcionDivisionEntreCero si intenta dividir entre cero
13    if ( denominador == 0 )
14       throw ExcepcionDivisionEntreCero(); // termina la función
15
16    // devuelve el resultado de la división
17    return static_cast< double >( numerador ) / denominador;
18 } // fin de la función cociente
19
20 int main()
21 {
22    int numero1; // numerador especificado por el usuario
23    int numero2; // denominador especificado por el usuario
24
25    cout << "Escriba dos enteros (fin de archivo para terminar): ";
26
27    // permite al usuario introducir dos enteros para la división
28    while ( cin >> numero1 >> numero2 )
29    {
30       // el bloque try contiene código que podría lanzar una excepción
31       // y código que no se debe ejecutar si ocurre una excepción
32       try
33       {
34          double resultado = cociente( numero1, numero2 );
35          cout << "El cociente es: " << resultado << endl;
36       } // fin de try
37       catch ( ExcepcionDivisionEntreCero &excepcionDivisionEntreCero )
38       {
```

Fig. 17.2 | Ejemplo que lanza excepciones al tratar de dividir entre cero (parte 1 de 2).

```
39          cout << "Ocurrio una excepcion: "
40             << excepcionDivisionEntreCero.what() << endl;
41       } // fin de catch
42
43       cout << "\nEscriba dos enteros (fin de archivo para terminar): ";
44    } // fin de while
45
46    cout << endl;
47 } // fin de main
```

```
Escriba dos enteros (fin de archivo para terminar): 100 7
El cociente es: 14.2857

Escriba dos enteros (fin de archivo para terminar): 100 0
Ocurrio una excepcion: intento de dividir entre cero

Escriba dos enteros (fin de archivo para terminar): ^Z
```

Fig. 17.2 | Ejemplo que lanza excepciones al tratar de dividir entre cero (parte 2 de 2).

Encerrar código en un bloque `try`

El programa empieza pidiendo al usuario que introduzca dos enteros. Estos enteros se introducen en la condición del ciclo `while` (línea 28). En la línea 34 se pasan los valores a la función `cociente` (líneas 10 a 18), la cual divide los enteros y devuelve un resultado, o **lanza una excepción** (es decir, indica que ocurrió un error) en un intento de dividir entre cero. El manejo de excepciones está orientado a situaciones en las que la función que detecta un error no puede manejarlo.

Como vimos en la sección 7.10, los bloques `try` permiten el manejo de excepciones al encerrar las instrucciones que podrían ocasionar excepciones, e instrucciones que se deberían omitir si ocurre una excepción. El bloque `try` en líneas 32 a 36 encierra la invocación de la función `cociente` y la instrucción que muestra el resultado de la división. En este ejemplo, debido a que la invocación de la función `cociente` (línea 34) puede *lanzar* una excepción, encerramos la invocación a esta función en un bloque `try`. Al encerrar la instrucción de salida (línea 35) en el bloque `try`, aseguramos que la salida *sólo* ocurrirá si la función `cociente` devuelve un resultado.

Observación de Ingeniería de Software 17.3

Las excepciones pueden surgir a través de código mencionado en forma explícita en un bloque `try`*, a través de las llamadas a otras funciones y a través de llamadas a funciones con muchos niveles de anidamiento, iniciadas por el código en un bloque* `try`*.*

Definición de un manejador `catch` *para procesar una* `ExcepcionDivisionEntreCero`

En la sección 7.10 vimos que las excepciones se procesan mediante los manejadores `catch`. Por lo menos *debe* haber un manejador `catch` (líneas 37 a 41) inmediatamente después de cada bloque `try`. Un parámetro de excepcion *siempre* debe declararse como una *referencia* al tipo de excepción que pueda procesar el manejador `catch` (en este caso, `ExcepcionDivisionEntreCero`); esto evita tener que copiar el objeto excepción al momento de atraparlo y permite que un manejador `catch` atrape correctamente las excepciones de clases derivadas también. Cuando ocurre una excepción en un bloque `try`, el manejador `catch` que se ejecuta es aquél cuyo tipo *coincide* con el tipo de la excepción que ocurrió (es decir, el tipo en el bloque `catch` coincide exactamente con el tipo de excepción lanzada, o es una clase base *directa* o *indirecta* de la misma). Si un parámetro de excepción incluye un nombre de parámetro *opcional*, el manejador `catch` puede usar ese nombre de parámetro para interactuar con la excepción atrapada en el cuerpo del manejador `catch`, que está delimitado por llaves (`{` y `}`). Por lo general, un manejador `catch` repor-

ta el error al usuario, lo registra en un archivo, termina el programa sin que haya pérdida de datos o intenta una estrategia alterna para realizar la tarea fallida. En este ejemplo, el manejador `catch` simplemente reporta que el usuario trató de realizar una división entre cero. Después, el programa pide al usuario que introduzca dos nuevos valores enteros.

Error común de programación 17.1

Es un error de sintaxis colocar código entre un bloque `try` *y sus correspondientes manejadores* `catch`*, o entre sus manejadores* `catch`*.*

Error común de programación 17.2

Cada manejador `catch` *puede tener un solo parámetro; es un error de sintaxis especificar una lista separada por comas de parámetros de excepciones.*

Error común de programación 17.3

Es un error de compilación atrapar el mismo *tipo en varios manejadores* `catch` *que vayan después de un solo bloque* `try`*.*

Modelo de terminación del manejo de excepciones

Si ocurre una excepción como resultado de una instrucción en un bloque `try`, este bloque expira (es decir, termina de inmediato). A continuación, el programa busca el primer manejador `catch` que pueda procesar el tipo de excepción que ocurrió. El programa localiza el `catch` que coincida, comparando el tipo de la excepción lanzada con el tipo del parámetro de excepción de cada `catch`, hasta que el programa encuentra una coincidencia. Ocurre una coincidencia si los tipos son *idénticos*, o si el tipo de la excepción lanzada es una *clase derivada* del tipo del parámetro de excepción. Cuando ocurre una coincidencia, se ejecuta el código contenido en el manejador `catch` que coincide. Cuando un manejador `catch` termina su procesamiento al llegar a su llave derecha de cierre (`}`), se considera que la excepción se manejó y las variables locales definidas dentro del manejador `catch` (incluyendo el parámetro de `catch`) quedan fuera de alcance. El control del programa *no* regresa al punto en el que ocurrió la excepción (conocido como el **punto de lanzamiento**), debido a que el bloque `try` ha *expirado*. En vez de ello, el control continúa con la primera instrucción (línea 43) después del último manejador `catch` que sigue del bloque `try`. A esto se le conoce como **modelo de terminación del manejo de excepciones**. Algunos lenguajes usan el **modelo de reanudación del manejo de excepciones**, en el que después de manejar una excepción, el control se reanuda justo después del punto de lanzamiento. Al igual que con cualquier otro bloque de código, *cuando termina un bloque* `try`*, las variables definidas en el bloque quedan fuera de alcance*.

Error común de programación 17.4

Pueden ocurrir errores lógicos si asumimos que, después de manejar una excepción, el control regresará a la primera instrucción que sigue después del punto de lanzamiento.

Tip para prevenir errores 17.2

Con el manejo de excepciones, un programa puede seguir ejecutándose (en vez de terminar) después de lidiar con un problema. Esto ayuda a asegurar el tipo de aplicaciones robustas que contribuyen a lo que se conoce como computación de misión crítica, o computación crítica para los negocios.

Si el bloque `try` completa su ejecución con éxito (es decir, si no ocurren excepciones en el bloque `try`), entonces el programa ignora los manejadores `catch` y el control del programa continúa con la primera instrucción después del último bloque `catch` que sigue de ese bloque `try`.

Si una excepción que ocurre en un bloque `try` *no* tiene un manejador `catch` que coincida, o si una excepción ocurre en una instrucción que *no* se encuentre dentro de un bloque `try`, la función que contiene la instrucción termina de inmediato y el programa intenta localizar un bloque `try` circundante en la función que hizo la llamada. A este proceso se le conoce como **limpieza de la pila** y se describe en la sección 17.4.

Flujo del control del programa cuando el usuario introduce un denominador distinto de cero

Considere el flujo de control cuando el usuario introduce el numerador `100` y el denominador `7`. En la línea 13, la función `cociente` determina que el `denominador` no es igual a cero, por lo que en la línea 17 se realiza la división y se devuelve el resultado (`14.2857`) a la línea 34 como un valor `double`. Después, el control del programa continúa secuencialmente desde la línea 34, por lo que en la línea 35 se muestra el resultado de la división; en la línea 36 se termina el bloque `try`. Como el bloque `try` se completó con éxito y *no* lanzó una excepción, el programa *no* ejecuta las instrucciones contenidas en el manejador `catch` (líneas 37 a 41), y el control continúa a la línea 43 (la primera línea de código después del manejador `catch`), en donde se pide al usuario que introduzca dos enteros más.

Flujo del control del programa cuando el usuario escribe un denominador de cero

Ahora vamos a considerar un caso en el que el usuario introduce el numerador `100` y el denominador `0`. En la línea 13, `cociente` determina que el `denominador` es igual a cero, lo cual indica un intento de división entre cero. En la línea 14 se lanza una excepción, que representamos como un objeto de la clase `ExcepcionDivisionEntreCero` (figura 17.1).

Para lanzar una excepción, en la línea 14 de la figura 17.2 se utiliza la palabra clave **`throw`** seguida de un operando del tipo de excepción a lanzar. Por lo general, una instrucción `throw` especifica *un* operando. (En la sección 17.3 veremos cómo usar una instrucción `throw` *sin* operandos). El operando de una instrucción `throw` puede ser de *cualquier* tipo (pero debe ser construible por copia). Si el operando es un objeto, lo llamamos **objeto excepción**; en este ejemplo, el objeto excepción es un objeto de tipo `ExcepcionDivisionEntreCero`. Sin embargo, un operando de `throw` también puede asumir otros valores, como el valor de una expresión que *no* produce un objeto de una clase (por ejemplo, `throw x > 5`) o el valor de un `int` (por ejemplo, `throw 5`). Los ejemplos en este capítulo se enfocan exclusivamente en cómo lanzar objetos de clases de excepciones.

Tip para prevenir errores 17.3

En general, sólo se deben lanzar objetos de tipos de clases de excepciones.

Como parte de lanzar una excepción, se crea el operando `throw` y se utiliza para inicializar el parámetro en el manejador `catch`, el cual veremos en breve. La instrucción `throw` en la línea 17 crea un objeto de la clase `ExcepcionDivisionEntreCero`. Cuando la línea 14 lanza la excepción, la función `cociente` sale de inmediato. Por lo tanto, en la línea 14 se lanza la excepción *antes* de que la función `cociente` pueda realizar la división en la línea 17. Esta es una característica central del manejo de excepciones: *si su programa lanza de manera explícita una excepción, debe hacerlo antes de que el error tenga la oportunidad de ocurrir*.

Puesto que encerramos la llamada a la función `cociente` (línea 34) en un bloque `try`, el control del programa entra al manejador `catch` (líneas 37 a 41) que está justo después del bloque `try`. Este manejador `catch` sirve como el manejador de excepciones para la excepción de división entre cero. En general, cuando se lanza una excepción dentro de un bloque `try`, la excepción se atrapa mediante un manejador `catch` que especifica el tipo que coincide con la excepción lanzada. En este programa, el manejador `catch` especifica que atrapa objetos `ExcepcionDivisionEntreCero`; este tipo coincide con el tipo del objeto lanzado en la función `cociente`. En realidad, el manejador `catch` atrapa una *referencia* al

objeto `ExcepcionDivisionEntreCero` creado por la instrucción `throw` de la función `cociente` (línea 14), de modo que el manejador `catch` *no* crea una copia del objeto excepción.

El cuerpo del `catch` (líneas 39 y 40) imprime el mensaje de error devuelto por la función `what` de la clase base `runtime_error`; es decir, la cadena que el constructor de `ExcepcionDivisionEntreCero` (líneas 11 y 12 de la figura 17.1) pasó al constructor de la clase base `runtime_error`.

Buena práctica de programación 17.1

Al asociar cada tipo de error en tiempo de ejecución con un tipo de excepción con nombre apropiado, se mejora la claridad del programa.

17.3 Volver a lanzar una excepción

Una función podría utilizar un recurso (como un archivo) y tal vez desee liberarlo (es decir, cerrar el archivo) en caso de que ocurra una excepción. Un manejador de excepciones, al momento de recibir una excepción, puede liberar el recurso y luego notificar a la función que lo llamó que ocurrió una excepción, para lo cual debe **volver a lanzar la excepción** mediante la siguiente instrucción:

```
throw;
```

Sin importar el que un manejador pueda o no procesar una excepción, el manejador puede *volver a lanzar* la excepción para seguirla procesando fuera de éste. El siguiente bloque `try` circundante detecta la excepción que se volvió a lanzar, y un manejador `catch` listado después de ese bloque `try` circundante trata de manejarla.

Error común de programación 17.5

Al ejecutar una instrucción `throw` vacía que se sitúa fuera de un manejador `catch` se abandona el procesamiento de la excepción y termina el programa de inmediato.

El programa de la figura 17.3 demuestra cómo volver a lanzar una excepción. En el bloque `try` de `main` (líneas 29 a 34), la línea 32 llama a la función `lanzarExcepcion` (líneas 8 a 24). Esta función también contiene un bloque `try` (líneas 11 a 15), desde el que la instrucción `throw` en la línea 14 lanza una instancia de la clase `exception` de la biblioteca estándar. El manejador `catch` de la función `lanzarExcepcion` (líneas 16 a 21) atrapa esta excepción, imprime un mensaje de error (líneas 18 y 19) y vuelve a lanzar la excepción (línea 20). Esto termina la función `lanzarExcepcion` y devuelve el control a la línea 32 en el bloque `try...catch` en `main`. El bloque `try` *termina* (por lo que la línea 33 *no* se ejecuta), y el manejador `catch` en `main` (líneas 35 a 38) atrapa esta excepción e imprime un mensaje de error (línea 37). Como no utilizamos los parámetros de excepción en los manejadores `catch` de este ejemplo, omitimos los nombres de los parámetros de excepción y especificamos sólo el tipo de excepción que se debe atrapar (líneas 16 y 35).

```
1  // Fig. 17.3: fig17_03.cpp
2  // Volver a lanzar una excepción.
3  #include <iostream>
4  #include <exception>
5  using namespace std;
6
```

Fig. 17.3 | Volver a lanzar una excepción (parte 1 de 2).

```
// lanza, atrapa y vuelve a lanzar la excepción
void lanzarExcepcion()
{
   // lanza la excepción y la atrapa de inmediato
   try
   {
      cout << "  la funcion lanzarExcepcion lanza una excepcion\n";
      throw excepcion(); // genera la excepción
   } // fin de try
   catch ( excepcion & ) // maneja la excepción
   {
      cout << "  La excepcion se manejo en la funcion lanzarExcepcion"
         << "\n  La funcion lanzarExcepcion vuelve a lanzar la excepcion";
      throw; // vuelve a lanzar la excepción para seguir procesándola
   } // fin de catch

   cout << "Esto tampoco se debe imprimir\n";
} // fin de la función lanzarExcepcion

int main()
{
   // lanza la excepción
   try
   {
      cout << "\nmain invoca a la funcion lanzarExcepcion\n";
      lanzarExcepcion();
      cout << "Esto no se debe imprimir\n";
   } // fin de try
   catch ( excepcion & ) // maneja la excepción
   {
      cout << "\n\nLa excepcion se manejo en main\n";
   } // fin de catch

   cout << "El control del programa continua despues de catch en main\n";
} // fin de main
```

```
main invoca a la funcion lanzarExcepcion
  La funcion lanzarExcepcion lanza una excepcion
  La excepcion se manejo en la funcion lanzarExcepcion
  La funcion lanzarExcepcion vuelve a lanzar la excepcion

La excepcion se manejo en main
El control del programa continua despues de catch en main
```

Fig. 17.3 | Volver a lanzar una excepción (parte 2 de 2).

17.4 Limpieza de la pila

Cuando se lanza una excepción pero no se atrapa en un alcance específico, la pila de llamadas a funciones se "limpia" y se hace un intento de atrapar (`catch`) la excepción en el siguiente bloque `try...catch` exterior. Limpiar la pila de llamadas a funciones significa que la función en la que no se atrapó la excepción termina, todas las variables locales que hayan completado su inicialización en esa función se des-

truyen y el control regresa a la instrucción que invocó originalmente a esa función. Si un bloque `try` encierra esa instrucción, se hace un intento de atrapar la excepción. Si un bloque `try` *no* encierra esa instrucción, se vuelve a realizar la limpieza de la pila. Si no hay un manejador `catch` que atrape a esta excepción, el programa termina. El programa de la figura 17.4 demuestra la limpieza de la pila.

```
1  // Fig. 17.4: fig17_04.cpp
2  // Demostración de la limpieza de la pila.
3  #include <iostream>
4  #include <stdexcept>
5  using namespace std;
6
7  // funcion3 lanza un error en tiempo de ejecución
8  void funcion3()
9  {
10    cout << "En la funcion 3" << endl;
11
12    // no hay bloque try, se realiza la limpieza de la pila, devuelve el control a
         funcion2
13    throw runtime_error( "runtime_error en funcion3" ); // no imprime
14 } // fin de funcion3
15
16 // funcion2 invoca a funcion3
17 void funcion2()
18 {
19    cout << "funcion3 se llama dentro de funcion2" << endl;
20    funcion3(); // se realiza la limpieza de la pila, devuelve el control a funcion1
21 } // fin de funcion2
22
23 // funcion1 invoca a funcion2
24 void funcion1()
25 {
26    cout << "funcion2 se llama dentro de funcion1" << endl;
27    funcion2(); // se realiza la limpieza de la pila, devuelve el control a main
28 } // fin de funcion1
29
30 // demuestra la limpieza de la pila
31 int main()
32 {
33    // invoca a funcion1
34    try
35    {
36       cout << "funcion1 se llama dentro de main" << endl;
37       funcion1(); // llama a funcion1, que lanza runtime_error
38    } // fin de try
39    catch ( runtime_error &error ) // maneja el error en tiempo de ejecución
40    {
41       cout << "Ocurrio una excepcion: "  << error.what() << endl;
42       cout << "La excepcion se manejo en main" << endl;
43    } // fin de catch
44 } // fin de main
```

Fig. 17.4 | Limpieza de la pila (parte 1 de 2).

```
funcion1 se llama dentro de main
funcion2 se llama dentro de funcion1
funcion3 se llama dentro de funcion2
En la funcion 3
Ocurrio una excepcion: runtime_error en funcion3
La excepcion se manejo en main
```

Fig. 17.4 | Limpieza de la pila (parte 2 de 2).

En `main`, el bloque `try` (líneas 34 a 38) llama a `funcion1` (líneas 24 a 28). A continuación, `funcion1` llama a `funcion2` (líneas 17 a 21), que a su vez llama a `funcion3` (líneas 8 a 14). En la línea 13 de `funcion3` se lanza un objeto `runtime_error`. Sin embargo, debido a que ningún bloque `try` encierra la instrucción `throw` en la línea 13, se realiza la limpieza de la pila; `funcion3` termina en la línea 13, y después devuelve el control a la instrucción en `funcion2` que invocó a `funcion3` (línea 20). Como no hay un bloque `try` que encierre la línea 20, se realiza la limpieza de la pila otra vez; `funcion2` termina en la línea 20 y devuelve el control a la instrucción en `funcion1` que invocó a `funcion2` (línea 27). Como no hay bloque `try` que encierre la línea 27, se realiza la limpieza de la pila una vez más; `funcion1` termina en la línea 27 y devuelve el control a la instrucción en `main` que invocó a `funcion1` (línea 37). El bloque `try` de las líneas 34 a 38 encierra esta instrucción, por lo que el primer manejador `catch` que coincide y que está ubicado después de este bloque `try` (líneas 39 a 43) atrapa y procesa la excepción. En la línea 41 se usa la función `what` para mostrar el mensaje de la excepción.

17.5 Cuándo utilizar el manejo de excepciones

El manejo de excepciones está diseñado para procesar **errores sincrónicos**, que ocurren cuando se ejecuta una instrucción, como *subíndices de arreglos fuera de rango*, *desbordamiento aritmético* (es decir, un valor fuera del rango de valores representables), *división entre cero*, *parámetros de función inválidos* y *asignación fallida de memoria* (debido a la falta de memoria). El manejo de excepciones no está diseñado para procesar los errores asociados con los **eventos sincrónicos** (por ejemplo, completar la E/S de disco, la llegada de mensajes de red, los clics del ratón y las pulsaciones de tecla), los cuales ocurren en paralelo con, y de manera independiente a, el flujo de control del programa.

Observación de Ingeniería de Software 17.4

El manejo de excepciones proporciona una sola técnica uniforme para procesar problemas. Esto ayuda a los programadores, que trabajan en proyectos extensos, a comprender el código de procesamiento de errores de los demás programadores.

Observación de Ingeniería de Software 17.5

El manejo de excepciones permite que los componentes de software predefinidos comuniquen los problemas a los componentes específicos de una aplicación, que a su vez pueden procesar los problemas en forma específica para la aplicación.

El manejo de excepciones también es útil para procesar problemas que ocurren cuando un programa interactúa con los elementos de software, como las funciones miembro, los constructores, los destructores y las clases. Dichos elementos de software utilizan comúnmente las excepciones para notificar a los programas cuando ocurren problemas. Esto permite a los programadores implementar el *manejo de errores personalizado* para cada aplicación.

Observación de Ingeniería de Software 17.16

Las funciones con condiciones de errores comunes deben devolver `nullptr`*,* `0` *u otros valores apropiados, como* `bool`*, en vez de lanzar excepciones. Un programa que llame a dicha función puede comprobar el valor de retorno para determinar si la llamada a la función tuvo éxito o fracasó.*

Por lo general, las aplicaciones complejas consisten en componentes predefinidos de software y componentes específicos de cada aplicación que utilizan los componentes predefinidos. Cuando un componente predefinido encuentra un problema, ese componente necesita un mecanismo para comunicar el problema al componente específico de la aplicación; el *componente predefinido no puede saber de antemano cómo procesa cada aplicación un problema que se presenta*.

C++11: declaración de funciones que no lanzan excepciones

A partir de C++11, si una función no lanza excepciones *y* no llama a funciones que lancen excepciones, debemos declarar de manera explícita que una función *no* lanza excepciones. Esto indica a los programadores del código cliente que no hay necesidad de colocar las llamadas a la función en un bloque `try`. Sólo hay que agregar **`noexcept`** a la derecha de la lista de parámetros de la función, tanto en el prototipo como en la definición. Para una función miembro `const`, hay que colocar `noexcept` después de `const`. Si una función declarada como `noexcept` llama a otra función que lance una excepción o ejecute una instrucción `throw`, el programa termina. En el capítulo 24 veremos más sobre `noexcept`.

17.6 Constructores, destructores y manejo de excepciones

Primero vamos a hablar sobre una cuestión que hemos mencionado, pero todavía no hemos resuelto de manera satisfactoria: ¿qué ocurre cuando se detecta un error en un *constructor*? Por ejemplo, ¿cómo debe responder el constructor de un objeto al recibir datos inválidos? Como el constructor *no puede devolver un valor* para indicar un error, debemos elegir un medio alternativo de indicar que el objeto no se ha construido en forma apropiada. Un esquema es devolver el objeto mal construido y esperar que alguien que lo use realice las pruebas apropiadas para determinar que se encuentra en un estado inconsistente. Otro esquema es establecer una variable fuera del constructor. La alternativa preferida es requerir que el constructor lance (`throw`) una excepción que contenga la información del error, con lo cual se ofrece una oportunidad para que el programa maneje la falla.

Antes de que un constructor lance una excepción, se hacen llamadas a los destructores para cualquier objeto miembro que se construya como parte del objeto que se va a construir. Se hacen llamadas a los destructores para todos los objetos automáticos construidos en un bloque `try` antes de atrapar la excepción. Se garantiza que la limpieza de la pila se habrá completado en el momento en el que un manejador de excepciones se empiece a ejecutar. Si un destructor invocado como resultado de la limpieza de la pila lanza una excepción, el programa termina. Esto se ha vinculado con varios ataques de seguridad.

Tip para prevenir errores 17.4

Los destructores deben atrapar excepciones para prevenir la terminación del programa.

Tip para prevenir errores 17.5

No lance excepciones desde el constructor de un objeto con duración de almacenamiento estática. Dichas excepciones no pueden atraparse.

Si un objeto tiene objetos miembro, y si se lanza una excepción antes de que el objeto exterior se construya por completo, entonces se ejecutarán los destructores para los objetos miembro que se hayan construido antes de la ocurrencia de la excepción. Si se ha construido parcialmente un arreglo de objetos cuando ocurre una excepción, sólo se llamará a los destructores para los objetos construidos en el arreglo.

Tip para prevenir errores 17.6

Cuando se lanza una excepción desde el constructor para un objeto que se cree en una expresión `new`*, se libera la memoria asignada en forma dinámica para ese objeto.*

Tip para prevenir errores 17.7

Un constructor debe lanzar una excepción si ocurre un problema al inicializar un objeto. Antes de hacerlo, el constructor debe liberar la memoria que haya asignado en forma dinámica.

Inicialización de objetos locales para adquirir recursos

Una excepción podría impedir la operación de código que por lo general *liberaría un recurso* (como memoria o un archivo), con lo cual se produciría una **fuga de recursos** que evitará que otros programas adquieran el recurso. Una técnica para resolver este problema es inicializar un objeto local para adquirir el recurso. Cuando ocurra una excepción, se invocará al destructor para ese objeto y se podrá liberar el recurso.

17.7 Excepciones y herencia

Se pueden derivar varias clases de excepciones de una clase base común, como vimos en la sección 17.2, cuando creamos la clase `ExcepcionDivisionEntreCero` como una clase derivada de la clase `exception`. Si un manejador `catch` maneja una referencia a un objeto excepción del tipo de una clase base, también puede atrapar una referencia a todos los objetos de las clases que se deriven públicamente de esa clase base; esto permite el procesamiento polimórfico de los errores relacionados.

Tip para prevenir errores 17.8

El uso de la herencia con excepciones permite a un manejador de excepciones atrapar (`catch`*) los errores relacionados con una notación concisa. Una metodología es atrapar cada tipo de apuntador o referencia a un objeto excepción de clase derivada en forma individual, pero una metodología más concisa es atrapar mejor apuntadores o referencias a objetos excepción de la clase base. Además, al atrapar de manera individual apuntadores o referencias a objetos excepción de una clase derivada se pueden cometer errores, en especial si el programador olvida evaluar de manera explícita uno o más de los tipos de referencias de la clase derivada.*

17.8 Procesamiento de las fallas de new

Cuando falla el operador `new`, lanza una excepción **`bad_alloc`** (definida en el encabezado `<new>`). En esta sección presentaremos dos ejemplos de fallas de `new`. El primer ejemplo utiliza la versión de `new` que lanza una excepción `bad_alloc` cuando falla `new`. El segundo utiliza la función **`set_new_handler`** para manejar las fallas de `new`. [*Nota:* los ejemplos de las figuras 17.5 y 17.6 asignan cantidades extensas de memoria dinámica, lo cual podría hacer que su computadora se vuelva lenta].

Caso en el que `new` *lanza* `bad_alloc` *al fallar*

La figura 17.5 demuestra cómo `new` lanza *implícitamente* `bad_alloc` al fallar en asignar la memoria solicitada. La instrucción `for` (líneas 16 a 20) dentro del bloque `try` debe iterar 50 veces y en cada pasada,

asigna un arreglo de 50,000,000 valores `double`. Si `new` falla y lanza una excepción `bad_alloc`, el ciclo termina y el programa continúa en la línea 22, en donde el manejador `catch` atrapa y procesa la excepción. En las líneas 24 y 25 se imprime el mensaje `"Ocurrio una excepcion:"` seguido del mensaje devuelto de la versión de la función `what` correspondiente a la clase base `exception` (es decir, un mensaje específico de la excepción, definido por la implementación, como `"bad allocation"` en Microsoft Visual C++). Los resultados muestran que el programa sólo realizó cuatro iteraciones del ciclo antes de que fallara `new` y se lanzara la excepción `bad_alloc`. Los resultados del lector podrían diferir con base en la memoria física, espacio en disco disponible para la memoria virtual en su sistema, y el compilador que esté utilizando.

```
1  // Fig. 17.5: fig17_05.cpp
2  // Demostración de new estándar que lanza una excepción
3  // bad_alloc cuando no se puede asignar memoria.
4  #include <iostream>
5  #include <new> // aquí se define la clase bad_alloc
6  using namespace std;
7
8  int main()
9  {
10    double *ptr[ 50 ];
11
12    // orienta cada ptr[i] a un bloque extenso de memoria
13    try
14    {
15       // asigna memoria para ptr[ i ]; new lanza bad_alloc al fallar
16       for ( size_t i = 0; i < 50; ++i )
17       {
18          ptr[ i ] = new double[ 50000000 ]; // puede lanzar una excepción
19          cout << "ptr[" << i << "] apunta a 50,000,000 nuevos valores double\n";
20       } // fin de for
21    } // fin de try
22    catch ( bad_alloc &excepcionAsignacionMemoria )
23    {
24       cerr << "Ocurrio una excepcion: "
25          << excepcionAsignacionMemoria.what() << endl;
26    } // fin de catch
27 } // fin de main
```

```
ptr[0] apunta a 50,000,000 nuevos valores double
ptr[1] apunta a 50,000,000 nuevos valores double
ptr[2] apunta a 50,000,000 nuevos valores double
ptr[3] apunta a 50,000,000 nuevos valores double
Ocurrio una excepcion: bad allocation
```

Fig. 17.5 | new lanza bad_alloc al fallar.

Caso en el que **new** *devuelve* **nullptr** *al fallar*

El estándar de C++ especifica que los programadores pueden usar una versión anterior de `new` que devuelva `nullptr` al fallar. Para este propósito, el encabezado `<new>` define el objeto **`nothrow`** (de tipo `nothrow_t`), que se utiliza de la siguiente manera:

```
double *ptr = new( nothrow ) double[ 50000000 ];
```

La instrucción anterior utiliza la versión de `new` que *no* lanza excepciones `bad_alloc` (es decir, `nothrow`) para asignar un arreglo de 50,000,000 valores `double`.

Observación de Ingeniería de Software 17.7

Para hacer los programas más robustos, utilice la versión de `new` que lanza excepciones `bad_alloc` al fallar.

Manejo de las fallas de `new` *mediante la función* `set_new_handler`

Una característica adicional para manejar las fallas de `new` es la función `set_new_handler` (cuyo prototipo se encuentra en el encabezado estándar `<new>`). Esta función recibe como argumento un apuntador a una función que no recibe argumentos y devuelve `void`. Este apuntador apunta a la función que se llamará si `new` falla. Esto proporciona al programador una metodología uniforme para manejar todas las fallas de `new`, sin importar que ocurra una falla en el programa. Una vez que `set_new_handler` registra un **manejador de new** en el programa, el operador `new` *no* lanza `bad_alloc` en el futuro; en vez de ello, difiere el manejo de errores a la función manejadora de `new`.

Si `new` asigna memoria con éxito, devuelve un apuntador a esa memoria. Si `new` falla en asignar memoria y `set_new_handler` no registró una función manejadora de `new`, `new` lanza una excepción `bad_alloc`. Si `new` falla al asignar memoria y se ha registrado una función manejadora de `new`, se hace una llamada a esta función. La función manejadora de `new` debe realizar una de las siguientes tareas:

1. Hacer más memoria disponible al eliminar otra parte de la memoria asignada en forma dinámica (o indicar al usuario que cierre otras aplicaciones) y regresar al operador `new` para tratar de asignar memoria otra vez.
2. Lanzar una excepción de tipo `bad_alloc`.
3. Llamar a la función `abort` o `exit` (ambas se encuentran en el encabezado `<cstdlib>`) para terminar el programa. Estas funciones se introdujeron en la sección 9.7.

La figura 17.6 demuestra el uso de `set_new_handler`. La función `nuevoManejadorPersonalizado` (líneas 9 a 13) imprime un mensaje de error (línea 11) y después termina el programa mediante una llamada a `abort` (línea 12). Los resultados muestran que el ciclo iteró cuatro veces antes de que `new` fallara e invocara a la función `nuevoManejadorPersonalizado`. Los resultados del lector podrían diferir con base en la memoria física, el espacio en disco disponible para la memoria virtual en su sistema y el compilador que utilice para compilar el programa.

```
1  // Fig. 17.6: fig17_06.cpp
2  // Demostración de set_new_handler.
3  #include <iostream>
4  #include <new> // prototipo de la función set_new_handler
5  #include <cstdlib> // prototipo de la función abort
6  using namespace std;
7
8  // maneja la falla de asignación de memoria
9  void nuevoManejadorPersonalizado()
10 {
11    cerr << "Se llamo a nuevoManejadorPersonalizado";
12    abort();
13 } // fin de la función nuevoManejadorPersonalizado
```

Fig. 17.6 | `set_new_handler` especifica la función a llamar cuando falla `new` (parte 1 de 2).

```
14
15  // uso de set_new_handler para manejar la asignación de memoria fallida
16  int main()
17  {
18     double *ptr[ 50 ];
19
20     // especifica que se debe llamar a nuevoManejadorPersonalizado
21     // al fallar la asignación de memoria
22     set_new_handler( nuevoManejadorPersonalizado );
23
24     // orienta cada ptr[i] a un bloque extenso de memoria; se llamará a
25     // nuevoManejadorPersonalizado al fallar la asignación de memoria
26     for ( size_t i = 0; i < 50; ++i )
27     {
28        ptr[ i ] = new double[ 50000000 ]; // puede lanzar una excepción
29        cout << "ptr[" << i << "] apunta a 50,000,000 nuevos valores double\n";
30     } // fin de for
31  } // fin de main
```

```
ptr[0] apunta a 50,000,000 nuevos valores double
ptr[1] apunta a 50,000,000 nuevos valores double
ptr[2] apunta a 50,000,000 nuevos valores double
ptr[3] apunta a 50,000,000 nuevos valores double
se llamo a nuevoManejadorPersonalizado
```

Fig. 17.6 | `set_new_handler` especifica la función a llamar cuando falla `new` (parte 2 de 2).

17.9 La clase unique_ptr y la asignación dinámica de memoria

Una práctica común de programación es *asignar* la memoria dinámica, asignar la dirección de la memoria a un apuntador, usar el apuntador para manipular la memoria y *desasignar* la memoria con `delete` cuando ésta ya no sea necesaria. Si ocurre una excepción después de una asignación exitosa de memoria, pero *antes* de que se ejecute la instrucción `delete`, podría ocurrir una *fuga de memoria*. C++ proporciona la plantilla de clase **`unique_ptr`** en el encabezado **`<memory>`** para lidiar con esta situación.

Un objeto de la clase `unique_ptr` mantiene un apuntador a la memoria que se asigna en forma dinámica. Cuando se hace una llamada al destructor de un objeto `unique_ptr` (por ejemplo, cuando un objeto `unique_ptr` queda fuera de alcance), realiza una operación `delete` con su miembro de datos apuntador. La plantilla de clase `unique_ptr` proporciona los operadores sobrecargados * y ->, de manera que un objeto `unique_ptr` se puede utilizar de la misma forma que una variable apuntador común. En la figura 17.9 se demuestra un objeto `unique_ptr` que apunta a un objeto de la clase `Entero` asignado en forma dinámica (figuras 17.7-17.8).

```
1  // Fig. 17.7: Entero.h
2  // Definición de la clase Entero.
3
4  class Entero
5  {
```

Fig. 17.7 | Definición de la clase `Entero` (parte 1 de 2).

```
6  public:
7     Entero( int i = 0 ); // constructor predeterminado de Entero
8     ~Entero(); // destructor de Entero
9     void establecerEntero( int i ); // establece el valor de un Entero
10    int obtenerEntero() const; // devuelve el valor de un Entero
11 private:
12    int valor;
13 }; // fin de la clase Entero
```

Fig. 17.7 | Definición de la clase `Entero` (parte 2 de 2).

```
1  // Fig. 17.8: Entero.cpp
2  // Definición de las funciones miembro de Entero.
3  #include <iostream>
4  #include "Entero.h"
5  using namespace std;
6
7  // constructor predeterminado de Entero
8  Entero::Entero( int i )
9     : valor( i )
10 {
11    cout << "Constructor para Entero " << valor << endl;
12 } // fin del constructor de Entero
13
14 // destructor de Entero
15 Entero::~Entero()
16 {
17    cout << "Destructor para Entero " << valor << endl;
18 } // fin del destructor de Entero
19
20 // establece el valor de Entero
21 void Entero::establecerEntero( int i )
22 {
23    valor = i;
24 } // fin de la función setEntero
25
26 // devuelve el valor de Entero
27 int Entero::obtenerEntero() const
28 {
29    return valor;
30 } // fin de la función obtenerEntero
```

Fig. 17.8 | Definiciones de las funciones miembro de la clase `Entero`.

En la línea 15 de la figura 17.9 se crea el objeto `ptrAEntero` de `unique_ptr`, y se inicializa con un apuntador a un objeto `Entero` asignado en forma dinámica, que contiene el valor `7`. En la línea 18 se utiliza el operador `->` sobrecargado de `unique_ptr` para invocar a la función `establecerEntero` en el objeto `Entero` que maneja `ptrAEntero`. En la línea 21 se utiliza el operador `*` sobrecargado de `unique_ptr` para desreferenciar a `ptrAEntero`, y después se utiliza el operador punto (`.`) para invocar a la función `obtenerEntero` en el objeto `Entero`. Al igual que un apuntador regular, los operadores sobrecargados `->` y `*` de un objeto `unique_ptr` se pueden utilizar para acceder al objeto al que apunta el objeto `unique_ptr`.

```
1  // Fig. 17.9: fig17_09.cpp
2  // Demostración de unique_ptr.
3  #include <iostream>
4  #include <memory>
5  using namespace std;
6
7  #include "Entero.h"
8
9  // usa unique_ptr para manipular un objeto Entero
10 int main()
11 {
12    cout << "Creacion de un objeto unique_ptr que apunta a un objeto Entero\n";
13
14    // "orienta" unique_ptr al objeto Entero
15    unique_ptr< Entero > ptrAEntero( new Entero( 7 ) );
16
17    cout << "\nUso de unique_ptr para manipular el objeto Entero\n";
18    ptrAEntero->establecerEntero( 99 ); // usa unique_ptr para establecer el valor del
      Entero value
19
20    // usa unique_ptr para obtener el valor del Entero
21    cout << "Entero despues de establecerEntero: "
      << ( *ptrAEntero ).obtenerEntero()
22       << "\n\nTerminando el programa" << endl;
23 } // fin de main
```

```
Creacion de un objeto auto_ptr que apunta a un objeto Entero
Constructor para Entero 7

Uso de auto_ptr para manipular el objeto Entero
Entero despues de setEntero: 99

Terminando el programa
Destructor para Entero 99
```

Fig. 17.9 | Un objeto `unique_ptr` maneja la memoria asignada en forma dinámica.

Como `ptrAEntero` es una variable automática local en `main`, se destruye cuando `main` termina. El destructor de `unique_ptr` obliga a que se lleve a cabo una operación `delete` del objeto `Entero` al que apunta `ptrAEntero`, que a su vez llama al destructor de la clase `Entero`. La memoria que ocupa `Entero` se libera, sin importar cómo sale el control del bloque (por ejemplo, mediante una instrucción `return` o mediante una excepción). Lo que es más importante, al utilizar esta técnica se pueden *evitar fugas de memoria*. Por ejemplo, suponga que una función devuelve un apuntador orientado a cierto objeto. Por desgracia, la función llamadora que recibe este apuntador tal vez no elimine el objeto mediante `delete`, con lo cual se produce una *fuga de memoria*. No obstante, si la función devuelve un objeto `unique_ptr` que apunte al objeto, éste se eliminará de manera automática cuando se haga la llamada al destructor del objeto `unique_ptr`.

Notas sobre `unique_ptr`

Esta clase se llama `unique_ptr` debido a que sólo *un* objeto `auto_ptr` puede poseer un objeto asignado en forma dinámica en un momento dado. Al usar su operador de asignación sobrecargado o su constructor de copia, un objeto `unique_ptr` puede *transferir la propiedad* de la memoria dinámica que maneja. El último objeto `unique_ptr` que mantiene el apuntador a la memoria dinámica eliminará

la memoria. Esto hace de `unique_ptr` un mecanismo ideal para devolver memoria asignada en forma dinámica al código cliente. Cuando el objeto `unique_ptr` queda fuera de alcance en el código *cliente*, el destructor de `unique_ptr` destruye el objeto asignado en forma dinámica y elimina su memoria.

`unique_ptr` a un arreglo integrado

También podemos usar un `unique_ptr` para manejar un arreglo integrado asignado en forma dinámica. Por ejemplo, considere la siguiente instrucción:

```
unique_ptr< string[] > ptr( new string[ 10 ] );
```

que asigna en forma dinámica un arreglo de 10 objetos `string` manejado por `ptr`. El tipo `string[]` indica que la memoria manejada es un arreglo integrado que contiene objetos `string`. Cuando un `unique_ptr` que maneja un arreglo queda fuera de alcance, elimina la memoria con `delete[]`, de modo que cada elemento del arreglo reciba una llamada al destructor.

Un `unique_ptr` que maneja un arreglo proporciona un operador `[]` sobrecargado para acceder a los elementos del arreglo. Por ejemplo, la instrucción

```
ptr[ 2 ] = "hola";
```

asigna `"hola"` al objeto `string en ptr[2]`, y la instrucción

```
cout << ptr[ 2 ] << endl;
```

muestra ese objeto `string`.

17.10 Jerarquía de excepciones de la Biblioteca estándar

La experiencia ha demostrado que las excepciones se pueden clasificar en varias categorías. La Biblioteca estándar de C++ incluye una jerarquía de clases de excepciones, algunas de las cuales se muestran en la figura 17.10. Como vimos por primera vez en la sección 17.2, esta jerarquía está encabezada por la clase base `exception` (definida en el encabezado `<exception>`), la cual contiene la función `virtual what` que las clases derivadas pueden sobrescribir para generar los mensajes de error apropiados.

Las clases derivadas inmediatas de la clase base `exception` incluyen a `runtime_error` y **`logic_error`** (ambas definidas en el encabezado `<stdexcept>`), cada una de las cuales tiene varias clases derivadas. De `exception` también se derivan las excepciones lanzadas por los operadores de C++; por ejemplo, `bad_alloc` es lanzada por `new` (sección 17.8), **`bad_cast`** es lanzada por `dynamic_cast` (capítulo 12) y **`bad_typeid`** es lanzada por `typeid` (capítulo 12).

Error común de programación 17.6

Colocar un manejador `catch` que atrape un objeto de la clase base antes de un `catch` que atrape un objeto de una clase derivada de esa clase base es un error lógico. El `catch` de la clase base atrapa todos los objetos de las clases derivadas de esa clase base, por lo que el `catch` de la clase derivada nunca se ejecutará.

La clase `logic_error` es la clase base de varias clases de excepciones estándar que indican errores en la lógica del programa. Por ejemplo, la clase **`invalid_argument`** indica que una función recibió un argumento inválido. (Sin duda, la codificación apropiada puede evitar que lleguen argumentos inválidos a una función). La clase **`length_error`** indica que para ese objeto se utilizó una longitud mayor que el tamaño máximo permitido para el objeto que se está manipulando. La clase **`out_of_range`** indica que un valor, como un subíndice en un arreglo, excedió su rango permitido de valores.

La clase `runtime_error`, que utilizamos brevemente en la sección 17.4, es la clase base de varias otras clases de excepciones estándar que indican errores en tiempo de ejecución. Por ejemplo, la clase

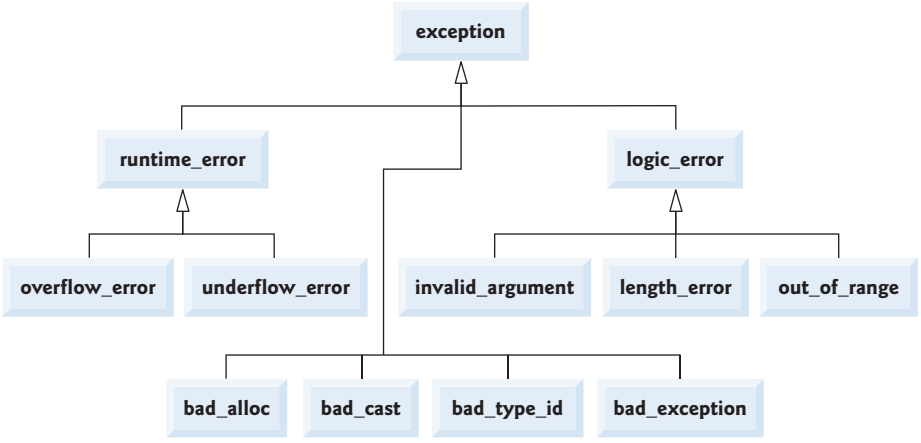

Fig. 17.10 | Algunas de las clases de excepciones de la Biblioteca estándar.

`overflow_error` describe un **error de desbordamiento aritmético** (es decir, el resultado de una operación aritmética es mayor que el número más grande que se puede almacenar en la computadora) y la clase `underflow_error` describe un **error de subdesbordamiento** (es decir, el resultado de una operación aritmética es menor que el número más pequeño que se puede almacenar en la computadora).

Error común de programación 17.7

Las clases de excepciones no necesitan derivarse de la clase `exception`*, por lo que atrapar el tipo* `exception` *no garantiza que se atraparán todas las excepciones que un programa pudiera encontrar.*

Tip para prevenir errores 17.9

Para atrapar todas las excepciones que se puedan llegar a lanzar en un bloque `try`*, use* `catch(...)`*. Una debilidad al atrapar las excepciones de esta forma es que se desconoce el tipo de las excepciones atrapadas. Otra debilidad es que, sin un parámetro con nombre, no hay forma de hacer referencia al objeto excepción dentro del manejador de la excepción.*

Observación de Ingeniería de Software 17.8

La jerarquía de `exception` *estándar es un buen punto de inicio para crear excepciones. Los programadores pueden crear programas que puedan lanzar excepciones estándar, excepciones derivadas de las excepciones estándar o sus propias excepciones que no se deriven de las excepciones estándar.*

Observación de Ingeniería de Software 17.9

Use `catch(...)` *para realizar una recuperación que no dependa del tipo de la excepción (por ejemplo, al liberar recursos comunes). La excepción se puede volver a lanzar para alertar a los manejadores* `catch` *circundantes más específicos.*

17.11 Conclusión

En este capítulo el lector aprendió a usar el manejo de excepciones para lidiar con los errores en un programa. Aprendió que el manejo de excepciones permite a los programadores eliminar el código de manejo de errores de la "línea principal" de ejecución del programa. Demostramos el manejo de excepciones

en el contexto de un ejemplo de división entre cero. Repasamos cómo usar bloques `try` para encerrar código que puede lanzar una excepción, y cómo usar manejadores `catch` para lidiar con las excepciones que puedan surgir. Aprendió a lanzar y volver a lanzar excepciones, y cómo manejar las excepciones que ocurren en los constructores. El capítulo continuó con discusiones acerca del procesamiento de las fallas de `new`, la asignación dinámica de memoria con la clase `unique_ptr` y la jerarquía de excepciones de la Biblioteca estándar. En el siguiente capítulo aprenderá a crear sus propias plantillas de clases personalizadas. En específico, demostraremos las herramientas que necesitará para crear sus propias estructuras de datos en plantillas personalizadas en el capítulo 19.

Resumen

Sección 17.1 Introducción

- Una excepción (pág. 741) es una indicación de un problema que ocurre durante la ejecución de un programa.
- El manejo de excepciones (pág. 741) nos permite crear programas que pueden resolver los problemas que ocurren en tiempo de ejecución; por lo general esto permite a los programas continuar su ejecución, como si no se hubiera encontrado ningún problema. Los problemas más graves pueden requerir que un programa notifique al usuario acerca del problema, antes de terminar de una manera controlada.

Sección 17.2 Ejemplo: manejo de un intento de dividir entre cero

- La clase `exception` es la clase base estándar de para las clases de excepciones (pág. 742). Esta clase cuenta con la función virtual `what` (pág. 742), que devuelve un mensaje de error apropiado y puede sobrescribirse en las clases derivadas.
- La clase `runtime_error` (pág. 742), que se define en el encabezado `<stdexcept>` (pág. 742), es la clase base estándar de C++ para representar los errores en tiempo de ejecución.
- C++ utiliza el modelo de terminación (pág. 745) del manejo de excepciones.
- Un bloque `try` consiste en la palabra clave `try`, seguida de llaves (`{}`) que definen un bloque de código en el que podrían ocurrir excepciones. El bloque `try` encierra instrucciones que podrían producir excepciones, e instrucciones que no deben ejecutarse si se producen excepciones.
- Por lo menos debe haber un manejador `catch` justo después de un bloque `try`. Cada manejador `catch` especifica un parámetro de excepción que representa el tipo de excepción que el manejador `catch` puede procesar.
- Si un parámetro de excepción incluye un nombre de parámetro opcional, el manejador `catch` puede usar ese nombre de parámetro para interactuar con un objeto excepción atrapado (pág. 746).
- El punto en el programa en el que ocurre una excepción se conoce como el punto de lanzamiento (pág. 745).
- Si ocurre una excepción en un bloque `try`, este bloque expira y el control del programa se transfiere al primer `catch` en el que coincida el tipo del parámetro de excepción con el de la excepción lanzada.
- Cuanto un bloque `try` termina, las variables locales definidas en el bloque quedan fuera de alcance.
- Cuando un bloque `try` termina debido a una excepción, el programa busca el primer manejador `catch` que coincida con el tipo de excepción que ocurrió. Ocurre una coincidencia si los tipos son idénticos, o si el tipo de la excepción lanzada es una clase derivada del tipo del parámetro de excepción. Cuando ocurre una coincidencia, se ejecuta el código contenido dentro del manejador `catch` que coincide.
- Cuando un manejador `catch` termina su procesamiento, el parámetro de `catch` y las variables locales definidas dentro del manejador `catch` quedan fuera de alcance. Cualquier manejador `catch` restante que corresponda al bloque `try` se ignora, y la ejecución se reanuda en la primera línea de código después de la secuencia `try...catch`.
- Si no ocurren excepciones en un bloque `try`, el programa ignora el (los) manejador(es) `catch` para ese bloque. La ejecución del programa se reanuda con la siguiente instrucción después de la secuencia `try...catch`.

- Si una excepción que ocurre en un bloque `try` no tiene un manejador `catch` que coincida, o si ocurre una excepción en una instrucción que no esté en un bloque `try`, la función que contiene la instrucción termina de inmediato y el programa trata de localizar un bloque `try` circundante en la función que hizo la llamada. A este proceso se le conoce como limpieza de la pila (pág. 746).
- Para lanzar una excepción, use la palabra clave `throw` seguida de un operando que representa el tipo de excepción a lanzar. El operando de una instrucción `throw` puede ser de cualquier tipo.

Sección 17.3 Volver a lanzar una excepción

- El manejador de excepciones puede diferir el manejo de una excepción (o tal vez una porción de ésta) a otro manejador de excepciones. En cualquier caso, para lograr esto el manejador vuelve a lanzar la excepción (pág. 747).
- Algunos ejemplos comunes de excepciones son los subíndices de arreglos fuera de rango, el desbordamiento aritmético, la división entre cero, los parámetros inválidos de funciones y las asignaciones de memoria fallidas.

Sección 17.4 Limpieza de la pila

- Limpiar la pila de llamadas a funciones significa que la función en la que la excepción no se atrapó termina, todas las variables locales en esa función se destruyen y el control regresa a la instrucción que invocó originalmente a esa función.

Sección 17.5 Cuándo utilizar el manejo de excepciones

- El manejo de excepciones es para errores sincrónicos (pág. 750), que ocurren cuando se ejecuta una instrucción.
- El manejo de excepciones no está diseñado para procesar los errores asociados con los eventos asíncronos (pág. 750), que ocurren en paralelo con el (y de manera independiente al) flujo de control del programa.
- A partir de C++11, si una función no lanza excepciones y no llama a funciones que lancen excepciones, debemos declarar explícitamente la función como `noexcept` (pág. 751).

Sección 17.6 Constructores, destructores y manejo de excepciones

- Las excepciones lanzadas por un constructor hacen que se llame a los destructores de todos los objetos creados como parte del objeto que se está construyendo, antes de que se lance la excepción.
- Cada objeto automático construido en un bloque `try` se destruye antes de lanzar una excepción.
- La limpieza de la pila se completa antes de que un manejador de excepciones empiece a ejecutarse.
- Si un destructor invocado como resultado de la limpieza de la pila lanza una excepción, el programa termina.
- Si un objeto tiene objetos miembros, y si se lanza una excepción antes de que el objeto exterior esté completamente construido, entonces se ejecutarán los destructores de los objetos miembro que se hayan construido antes de que ocurra la excepción.
- Si se ha construido parcialmente un arreglo de objetos cuando ocurre una excepción, sólo se llamará a los destructores para los objetos elementos del arreglo que estén construidos.
- Cuando se lanza una excepción desde el constructor para un objeto que se crea en una expresión `new`, se libera la memoria asignada en forma dinámica para ese objeto.

Sección 17.7 Excepciones y herencia

- Si un manejador `catch` atrapa una referencia a un objeto excepción del tipo de una clase base, también puede atrapar una referencia a todos los objetos de las clases que se deriven públicamente de esa clase base; esto permite el procesamiento polimórfico de los errores relacionados.

Sección 17.8 Procesamiento de las fallas de `new`

- El documento del estándar de C++ especifica que, cuando falla el operador `new`, lanza una excepción `bad_alloc` (pág, 752), que está definida en el encabezado `<new>`.
- La función `set_new_handler` (pág. 752) recibe como argumento un apuntador a una función que no recibe argumentos y devuelve `void`. Este apuntador apunta a la función que se llamará en caso de que falle `new`.

- Una vez que `set_new_handler` registra un manejador de `new` (pág. 754) en el programa, el operador `new` no lanza `bad_alloc` al fallar; en vez de ello difiere el manejo del error a la función manejadora de `new`.
- Si `new` asigna memoria con éxito, devuelve un apuntador a esa memoria.

Sección 17.9 La clase `unique_ptr` y la asignación dinámica de memoria

- Si ocurre una excepción después de la asignación exitosa de memoria pero antes de que se ejecute la instrucción `delete`, podría ocurrir una fuga de memoria.
- La Biblioteca estándar de C++ proporciona la plantilla de clase `unique_ptr` (pág. 755) para lidiar con las fugas de memoria.
- Un objeto de la clase `unique_ptr` mantiene un apuntador a la memoria asignada en forma dinámica. El destructor de un objeto `unique_ptr` realiza una operación `delete` en el miembro de datos apuntador del objeto `unique_ptr`.
- La plantilla de clase `unique_ptr` proporciona los operadores sobrecargados `*` y `->`, para que se pueda utilizar un objeto `unique_ptr` de la misma manera que una variable apuntador ordinaria. Un objeto `unique_ptr` también transfiere la propiedad de la memoria dinámica que maneja a través de su constructor de copia y su operador de asignación sobrecargado.

Sección 17.10 Jerarquía de excepciones de la Biblioteca estándar

- La Biblioteca estándar de C++ incluye una jerarquía de clases de excepciones. Esta jerarquía está encabezada por la clase base `exception`.
- Las clases derivadas inmediatas de la clase base `exception` son `runtime_error` y `logic_error` (ambas definidas en el encabezado `<stdexcept>`), cada una de las cuales tiene varias clases derivadas.
- Varios operadores lanzan excepciones estándar; el operador `new` lanza `bad_alloc`, el operador `dynamic_cast` lanza `bad_cast` (pág. 758) y el operador `typeid` lanza `bad_typeid` (pág. 758).

Ejercicios de autoevaluación

17.1 Liste cinco ejemplos comunes de excepciones.

17.2 De varias razones por las que las técnicas de manejo de excepciones no se deben utilizar para el control convencional de un programa.

17.3 ¿Por qué son las excepciones apropiadas para lidiar con los errores producidos por las funciones de biblioteca?

17.4 ¿Qué es una "fuga de recursos"?

17.5 Si no se lanzan excepciones en un bloque `try`, ¿a dónde procede el control después de que el bloque `try` termina de ejecutarse?

17.6 ¿Qué ocurre si se lanza una excepción fuera de un bloque `try`?

17.7 De una ventaja clave y una desventaja clave de utilizar `catch(...)`.

17.8 ¿Qué ocurre si ningún manejador `catch` coincide con el tipo de un objeto lanzado?

17.9 ¿Qué ocurre si varios manejadores coinciden con el tipo del objeto lanzado?

17.10 ¿Por qué un programador debe especificar un tipo de clase base como el tipo de un manejador `catch`, y después lanzar (`throw`) objetos de tipos de clases derivadas?

17.11 Suponga que está disponible un manejador `catch` que coincide exactamente con el tipo de un objeto excepción. ¿Bajo qué circunstancias podría ejecutarse un manejador diferente para los objetos excepción de ese tipo?

17.12 ¿Al lanzar una excepción se debe terminar el programa?

17.13 ¿Qué ocurre cuando un manejador `catch` lanza una excepción?

17.14 ¿Qué hace la instrucción `throw`?

Respuestas a los ejercicios de autoevaluación

17.1 Memoria insuficiente para satisfacer una petición de `new`, subíndice de arreglo fuera de límites, desbordamiento aritmético, división entre cero, parámetros de función inválidos.

17.2 (a) El manejo de excepciones está diseñado para manejar las situaciones que ocurren con poca frecuencia, y que a menudo provocan que el programa termine, de manera que los escritores de los compiladores no tengan que implementar el manejo de excepciones para obtener un rendimiento óptimo. (b) Por lo general, el flujo de control con las estructuras de control convencionales es más claro y eficiente que con las excepciones. (c) Pueden ocurrir problemas debido a que la pila se limpia cuando ocurre una excepción, y tal vez no se liberen los recursos asignados antes de la excepción. (d) Las excepciones "adicionales" hacen que sea más difícil para el programador manejar el número más grande de casos de excepciones.

17.3 Es improbable que una función de biblioteca realice el procesamiento de errores que cumpla con las necesidades específicas de cada usuario.

17.4 Un programa que termina en forma abrupta podría dejar un recurso en un estado en el que otros programas no puedan adquirirlo, o el programa en sí no podría readquirir un recurso "en fuga".

17.5 Los manejadores de excepciones (en los manejadores `catch`) para ese bloque `try` se omiten, y el programa reanuda su ejecución después del último manejador `catch`.

17.6 Una excepción lanzada fuera de un bloque `try` provoca una llamada a `terminate`.

17.7 La forma `catch(...)` atrapa cualquier tipo de excepción lanzada en un bloque `try`. Una ventaja es que se atraparán todas las posibles excepciones. Una desventaja es que `catch` no tiene parámetro, por lo que no puede hacer referencia a la información en el objeto lanzado y no puede conocer la causa de la excepción.

17.8 Esto hace que la búsqueda de una coincidencia continúe en el siguiente bloque `try` circundante, si hay uno. A medida que continúa este proceso, podría determinarse en un momento dado que no hay un manejador en el programa que coincida con el tipo del objeto lanzado; en este caso se termina el programa.

17.9 El primer manejador de excepciones que coincida después del bloque `try` se ejecuta.

17.10 Ésa es una excelente manera de atrapar tipos relacionados de excepciones.

17.11 Un manejador de la clase base atraparía objetos de todos los tipos de clases derivadas.

17.12 No, pero termina el bloque en el que se lanza la excepción.

17.13 La excepción será procesada por un manejador `catch` (si existe) asociado con el bloque `try` (si existe) que encierra al manejador `catch` que provocó la excepción.

17.14 Vuelve a lanzar la excepción si aparece en un manejador `catch`; en caso contrario, el programa termina.

Ejercicios

17.15 ***(Condiciones excepcionales)*** Liste varias condiciones excepcionales que han ocurrido a lo largo del libro. Liste todas las condiciones excepcionales adicionales que pueda. Para cada una de estas excepciones, describa brevemente cómo manejaría generalmente un programa la excepción, usando las técnicas para manejo de excepciones descritas en este capítulo. Algunas excepciones comunes son la división entre cero, el desbordamiento aritmético, el subíndice de un arreglo fuera de límites, agotamiento del almacén de memoria libre, etcétera.

17.16 ***(Parámetro de `catch`)*** ¿Bajo qué circunstancias no se debe proporcionar un nombre de parámetro al definir el tipo del objeto que será atrapado por un manejador?

17.17 ***(Instrucción `throw`)*** Un programa contiene la siguiente instrucción:

```
throw;
```

¿En dónde se esperaría encontrar comúnmente dicha instrucción? ¿Qué pasaría si esa instrucción apareciera en una parte distinta del programa?

17.18 ***(Manejo de excepciones* vs. *otros esquemas)*** Compare y contraste el manejo de excepciones con los otros esquemas de procesamiento de errores descritos en el texto.

17.19 ***(Manejo de excepciones y control del programa)*** ¿Por qué *no* deben usarse las excepciones como una forma alterna de control del programa?

17.20 *(Manejo de excepciones relacionadas)* Describa una técnica para manejar las excepciones relacionadas.

17.21 *(Lanzar excepciones desde un `catch`)* Suponga que un programa lanza una excepción y que se empieza a ejecutar el manejador de excepciones apropiado. Ahora suponga que el manejador de excepciones en sí lanza la misma excepción. ¿Crea esto una recursividad infinita? Escriba un programa para comprobar su observación.

17.22 *(Atrapar excepciones de clases derivadas)* Use la herencia para crear varias clases derivadas de `runtime_error`. Después muestre que un manejador `catch` que especifique la clase base puede atrapar excepciones de las clases derivadas.

17.23 *(Lanzar el resultado de una expresión condicional)* Lance el resultado de una expresión condicional que devuelva un valor `double` o un `int`. Proporcione un manejador `catch int` y un manejador `catch double`. Muestre que sólo se ejecuta el manejador `catch double`, sin importar que se devuelva el valor `int` o `double`.

17.24 *(Destructores de variables locales)* Escriba un programa que ilustre que todos los destructores para los objetos construidos en un bloque se llaman antes de que se lance una excepción desde ese bloque.

17.25 *(Destructores de objetos miembro)* Escriba un programa que ilustre que se llama a los destructores de los objetos miembro, sólo para aquellos objetos miembro que se construyeron antes de que ocurriera una excepción.

17.26 *(Atrapar todas las excepciones)* Escriba un programa que demuestre cómo se atrapan varios tipos de excepciones con el manejador de excepciones `catch(...)`.

17.27 *(Orden de los manejadores de excepciones)* Escriba un programa que ilustre que el orden de los manejadores de excepciones es importante. El primer manejador que coincida es el que se ejecuta. Trate de compilar y ejecutar su programa de dos maneras distintas, para mostrar que se ejecutan dos manejadores distintos con dos efectos diferentes.

17.28 *(Constructores que lanzan excepciones)* Escriba un programa que muestre cómo un constructor pasa información acerca de la falla del constructor a un manejador de excepciones después de un bloque `try`.

17.29 *(Volver a lanzar excepciones)* Escriba un programa que ilustre cómo volver a lanzar una excepción.

17.30 *(Excepciones no atrapadas)* Escriba un programa que ilustre que una función con su propio bloque `try` no tiene que atrapar todos los posibles errores generados dentro del bloque `try`. Algunas excepciones se pueden escabullir hasta (y ser manejadas en) los alcances exteriores.

17.31 *(Limpieza de pila)* Escriba un programa que lance una excepción desde una función con muchos niveles de anidamiento, y que de todas formas el manejador `catch` que sigue después del bloque `try` que encierra la llamada inicial en `main` atrape la excepción.