grandes variaciones a lo largo de una secuencia de operaciones. En este caso un análisis en el peor caso podría ser excesivamente pesimista y un análisis en el promedio, poco significativo, por la gran desviación de los tiempos.

El tiempo amortizado de un algoritmo se define como el cociente entre el tiempo total de una secuencia de ejecuciones del algoritmo y la longitud de dicha secuencia. Su empleo es apropiado cuando el tiempo de una secuencia tal es notablemente inferior al producto de su longitud por el tiempo en el peor caso del algoritmo.

Por ejemplo, el tiempo en el peor caso[2] para insertar un elemento al final de un vector extensible (*vector*) pertenece a $O(n)$, siendo n la longitud del vector, ya que si no queda espacio adicional sin utilizar, la operación de inserción tendrá que reservar un nuevo espacio de memoria y mover todos los elementos a la nueva zona. Sin embargo, cuando hay que expandir un vector de longitud n no se reserva espacio para un único elemento adicional, sino para $2n$. Esto permite que las siguientes $n-1$ inserciones no necesiten expandir el vector.

Esto implica que las $n-1$ inserciones posteriores tienen tiempos constantes, con lo que el tiempo total de una secuencia de n inserciones pertenece a $O(n)$. Amortizando este tiempo sobre las n inserciones se obtiene $O(1)$; se dice, pues, que la inserción al final de un vector es una operación de *tiempo amortizado constante*.

En adelante, para abreviar, emplearemos O_A cuando nos refiramos a un orden asintótico proveniente de un análisis amortizado. También emplearemos n para designar el número de elementos que intervienen en una operación, es decir, el tamaño de la entrada.

7.2. Contenedores e iteradores

7.2.1. Contenedores

Un *contenedor* es un objeto que se utiliza para almacenar otros objetos proporcionando operaciones para su manipulación. Como se verá, STL posee diversas clases contenedoras agrupadas en dos categorías: secuencias y contenedores asociativos ordenados. Los nombres de las clases y sus cabeceras estándares aparecen en la tabla 7.1.

Todos los contenedores poseen una serie de características comunes:

1. Definiciones públicas de tipos (véase la tabla 7.2).

[2] Contado en número de asignaciones de elementos.

Nombre	Cabecera	Descripción
vector	`<vector>`	Vectores
deque	`<deque>`	Colas dobles
list	`<list>`	Listas
set, *multiset*	`<set>`	Conjuntos y multiconjuntos
map, *multimap*	`<map>`	Aplicaciones mono y multivalor

Cuadro 7.1: Contenedores de la STL

2. Constructor por omisión que construye un contenedor vacío.
3. Constructor de copia y operador de asignación definidos con la semántica habitual.
4. Destructor.
5. Funciones miembro de tamaño (véase la tabla 7.3).
6. Operador `==` de igualdad estructural y operador `<` de comparación lexicográfica estricta.
7. Operadores relacionales adicionales (`!=`, `>`, `<=` y `>=`), definidos con la semántica habitual en función de `==` y `<`.
8. Funciones miembro de construcción de iteradores (véase la tabla 7.4).
9. Función miembro *swap()* que intercambia un contenedor por otro (normalmente en tiempo constante).

7.2.2. Iteradores

Los contenedores proporcionan *iteradores* para que a través de éstos sea posible recorrer sus elementos en un orden prefijado. Un iterador es una abstracción del concepto de puntero; de hecho, los punteros pueden emplearse en cualquier contexto en el que se requiera un iterador.

Los iteradores permiten separar los contenedores de los algoritmos genéricos. STL proporciona un rico conjunto de algoritmos genéricos que no trabajan directamente con los contenedores, sino con los iteradores proporcionados por dichos contenedores. Esto hace posible lo siguiente:

Tipo	Descripción
value_type	Tipo de los elementos del contenedor
reference	Tipo apropiado para almacenar elementos en el contenedor.
const_reference	Tipo apropiado para almacenar elementos constantes en el contenedor.
pointer	Tipo apropiado para apuntar a elementos del contenedor.
const_pointer	Tipo apropiado para apuntar a elementos constantes del contenedor.
iterator	Tipo apropiado para recorrer los elementos del contenedor.
const_iterator	Tipo apropiado para recorrer los elementos del contenedor, si ha sido definido constante.
reverse_iterator	Tipo apropiado para recorrer los elementos del contenedor en orden inverso.
const_reverse_iterator	Tipo apropiado para recorrer los elementos del contenedor en orden inverso, si ha sido definido constante.
difference_type	Tipo entero con signo, apropiado para representar la diferencia entre dos iteradores.
size_type	Tipo entero sin signo, apropiado para representar el tamaño del contenedor.

Cuadro 7.2: Miembros de tipo

Nombre	Complejidad	Descripción
empty()	$O_A(1)$	¿Está vacío el contenedor?
size()	$O(n)$	Tamaño (número de elementos) del contenedor.
max_size()	$O_A(1)$	Cota superior del tamaño del contenedor.

Cuadro 7.3: Funciones miembro de tamaño

Nombre	Complejidad	Descripción
begin()	$O_A(1)$	Devuelve un iterador apropiado para comenzar a recorrer el contenedor.
end()	$O_A(1)$	Devuelve un iterador apropiado para comprobar el final de un recorrido.
rbegin()	$O_A(1)$	Devuelve un iterador apropiado para comenzar a recorrer el contenedor en orden inverso.
rend()	$O_A(1)$	Devuelve un iterador apropiado para comprobar el final de un recorrido en orden inverso.

Cuadro 7.4: Funciones miembro de construcción de iteradores

1. Que un mismo algoritmo pueda aplicarse a distintos contenedores estándar, siempre que los iteradores que proporcionen sean «compatibles»[3] con los requeridos por el algoritmo.

2. Que un algoritmo pueda aplicarse a nuevos contenedores definidos por el usuario, siempre que éstos proporcionen iteradores que cumplan unos requisitos prefijados.

3. Que un algoritmo pueda aplicarse a tipos primitivos del lenguaje, puesto que los punteros son una especie de iteradores.

Las clases contenedoras permiten obtener iteradores adecuados a sus características particulares mediante las funciones de construcción de iteradores de la tabla 7.4.

Cada una de estas funciones de construcción de iteradores posee dos versiones, una normal y otra que es una sobrecarga `const` de ésta. Las funciones *begin()* y *end()* devuelven iteradores ordinarios (*iterator* o *const_iterator*) mientras que, en cambio, *rbegin()* y *rend()* devuelven iteradores inversos (*reverse_iterator* o *const_reverse_iterator*).

Dados dos iteradores *principio* y *fin*, se define su *rango* asociado, que se denota por $[principio, fin)$, como la secuencia de valores que comienza en *principio* y llega hasta *fin* sin incluirlo. Un rango cuyos extremos son iguales se denomina *rango vacío*.

[3]Esta compatibilidad se ha fijado dentro de STL atendiendo a razones de eficiencia; un algoritmo genérico no funcionará con un iterador del cual necesite operaciones cuya complejidad intrínseca aumente más allá de lo razonable la del propio algoritmo.

Las funciones *end()* y *rend()* siempre indican *una posición más allá* del elemento final del recorrido; por lo tanto, para recorrer un contenedor c completamente hay que iterar sobre el rango [`c.begin()`,`c.end()`).

Las clases y operaciones relacionadas con los iteradores se encuentran en `<iterator>`, que a su vez se incluye en muchos otros lugares.

Ejemplo:

Veamos diversas formas de recorrer un vector, por ejemplo, para mostrarlo en un flujo de salida. Para ello sobrecargaremos paramétricamente *operator* `<<` para que trabaje con vectores.

La primera consiste en acceder secuencialmente a cada uno de los elementos del vector mediante el operador de índice.

```
template <typename T>
ostream& operator <<(ostream& fs, const vector<T>& v)
{
  fs << "[ ";
  for (vector<T>::size_type i = 0; i < v.size(); ++i)
    fs << v[i] << ' ';
  return fs << ']';
}
```

Nótese cómo *vector<T>::size_type* es el tipo apropiado para representar un índice válido dentro del vector. De hecho, éste es el tipo devuelto por *vector<T>::size()*.

Pero existe otra opción: emplear iteradores. Para recorrer el vector completo hay que iterar sobre el rango [`v.begin()`,`v.end()`).

```
template <typename T>
ostream& operator <<(ostream& fs, const vector<T>& v)
{
  fs << "[ ";
  for (vector<T>::const_iterator i = v.begin(); i != v.end(); ++i)
    fs << *i << ' ';
  return fs << ']';
}
```

Observe que tanto *vector<T>::begin()* como *vector<T>::end()*, cuando se aplican a v, devuelven un *vector<T>::const_iterator*, ya que v es `const`.

Supongamos ahora que quisiéramos mostrar el vector «al revés». Al intentar modificar la versión con operador de índice tropezamos con un pequeño inconveniente, ya que

```
for (vector<T>::size_type i = v.size() - 1; i >= 0; --i)
  fs << v[i] << ' ';
```

es un bucle infinito. Aunque esto es fácil de arreglar de múltiples formas, la versión con iteradores se obtiene directamente empleando iteradores inversos:

```
for (vector<T>::const_reverse_iterator i = v.rbegin();
     i != v.rend(); ++i)
  fs << *i << ' ';
```

Es decir, para recorrer el vector completo «al revés» hay que iterar sobre el rango [`v.rbegin()`,`v.rend()`).

Además, las versiones con iteradores son ligeramente más eficientes en la práctica que las que emplean el operador de índice, por lo que se prefieren normalmente. Esto es similar a lo que ocurre en C, donde se puede recorrer un vector empleando punteros en vez del operador de índice (el cálculo de la dirección es, a nivel de máquina, algo más costoso que el incremento de un puntero).

Clasificación

Todos los tipos de iteradores definidos en STL disponen de sobrecargas de *operator* `*` y *operator* `++`, dividiéndose en diversas categorías dependiendo de las operaciones adicionales que se pueden realizar sobre ellos.

Iteradores de entrada Son aquéllos que permiten realizar un recorrido en una sola dirección leyendo el elemento apuntado. Sobrecargan también *operator* `->`.

Iteradores de salida Son aquéllos que permiten realizar un recorrido en una sola dirección escribiendo sobre el elemento apuntado.

Iteradores monodireccionales Son aquéllos que permiten realizar un recorrido en una sola dirección leyendo o escribiendo sobre el elemento apuntado. Sobrecargan también *operator* `->`.

Iteradores bidireccionales Son aquéllos que permiten realizar un recorrido en ambas direcciones leyendo o escribiendo sobre el elemento apuntado. Sobrecargan también *operator* `->` y *operator* `--`.

Iteradores de acceso directo Son aquéllos que permiten todas las operaciones que se pueden realizar con un puntero ordinario.

Todos los iteradores, a excepción de los de salida, poseen sobrecargas de *operator* `==` y *operator* `!=`.

Como se observa, los iteradores de entrada no garantizan que se pueda modificar el resultado de una operación `*` o `->`, mientras que los de salida sí garantizan la modificación a través de `*`, pero no la lectura.

Nótese también que los iteradores monodireccionales, bidireccionales y de acceso directo son *a la vez* iteradores de entrada y de salida.

Cada contenedor proporciona los iteradores apropiados para que puedan emplearse algoritmos eficientes sobre ellos. Los vectores y las colas dobles proporcionan iteradores de acceso directo[4]. Sin embargo, los iteradores de las listas y los contenedores asociativos son únicamente bidireccionales, ya que no es posible realizar acceso directo a un elemento de estos contenedores en tiempo constante[5].

Así, un algoritmo genérico diseñado eficientemente para emplear acceso directo, como es el caso del algoritmo de ordenación *sort()* de STL, sería ineficiente si se aplicara a una lista que proporcionara iteradores de «acceso directo» de complejidad $O(n)$. Por lo tanto, las listas no proporcionan tales iteradores y para compensar poseen una función miembro *sort()* especial.

Ejemplo:

El algoritmo genérico *copy()* recibe tres iteradores como parámetros. Los dos primeros son de entrada y el último es de salida. Esto permite el empleo de este algoritmo para transferir datos entre dos contenedores cualesquiera.

Iteradores de inserción

Si se escribe sobre un contenedor a través de un iterador de salida sin precaución, se estarán sobreescribiendo sus elementos. A menudo no es esto lo que se desea y suele ser causa de errores por desbordamiento del contenedor.

Ejemplo:

```
vector<double> v(10, 1.0), w(5, 2.0);
copy(v.begin(), v.end(), w.begin()); // Mal, «w» no tiene espacio
                                     // suficiente reservado
```

[4]El término «acceso directo» significa que se puede acceder a cualquier elemento en tiempo constante, es decir, que se puede considerar como una operación elemental.

[5]Dicho de otro modo, el acceso a un elemento no sería una operación elemental.

Más bien, lo que se desea en muchas ocasiones es algún tipo de inserción en el contenedor. Para lograr esto, STL define tres clases paramétricas denominadas *iteradores de inserción*; por cada una de éstas se define también una función de inserción o *insertor* que produce el iterador adecuado:

Insertores delanteros Se emplean para insertar elementos al principio de un contenedor que reciben como parámetro. Su nombre de función es *front_inserter()* y generan un objeto de tipo *front_insert_iterator*.

Insertores traseros Se emplean para insertar elementos al final de un contenedor que reciben como parámetro. Su nombre es *back_inserter()* y generan un *back_insert_iterator*.

Insertores «in situ» Se emplean para insertar elementos en un contenedor, que recibe como primer parámetro, en la posición anterior a la dada por un iterador, que recibe como segundo parámetro. Su nombre es *inserter()* y generan un *insert_iterator*.

Ejemplo:

```
vector<double> v(10, 1.0);
list<double> w(5, 2.0);
copy(v.begin(), v.end(), front_inserter(w)); // ins. al principio
copy(v.begin(), v.end(), back_inserter(w));  // ins. al final
```

Iteradores de flujo

STL proporciona clases que permiten tratar los flujos de E/S como si fueran secuencias. A estas clases se las denomina genéricamente *iteradores de flujo*.

La clase paramétrica *ostream_iterator* permite construir iteradores de salida capaces de escribir sobre un flujo de salida. Al construir un objeto de este tipo se le asocia un flujo de salida y, opcionalmente, una cadena de bajo nivel que actúa como terminador insertándose en el flujo automáticamente tras cada elemento.

Ejemplo:

Veamos otra forma de sobrecargar paramétricamente *operator* << para mostrar un vector en un flujo de salida.

```
template <typename T>
ostream& operator <<(ostream& fs, const vector<T>& v)
{
  fs << "[ ";
  copy(v.begin(), v.end(), ostream_iterator<T>(fs, " "));
  return fs << ']';
}
```

El objeto *ostream_iterator* permite insertar valores de tipo T separados por `" "` en el flujo de salida *fs*.

La clase paramétrica *istream_iterator* permite construir iteradores de entrada capaces de leer de un flujo de entrada. Al construir un objeto de este tipo se le puede asociar un flujo de entrada; no obstante, existe un constructor por omisión que crea un objeto apropiado para representar el fin de la entrada.

Ejemplo:

El siguiente fragmento lee números enteros de la entrada estándar separados por espacio blanco y los va insertando al final de un vector.

```
vector<int> v;
istream_iterator<int> ife(cin), // iterador de flujo de entrada
                      fin;      // fin de flujo de entrada
copy(ife, fin, back_inserter(v));
```

Observe cómo *fin* no lleva asociado ningún flujo concreto.

7.3. Secuencias

Una *secuencia* es un contenedor de elementos sobre los que no existe, a priori o de manera natural, un orden prefijado, y que presenta al exterior como si estuvieran dispuestos linealmente. Se definen tres tipos de secuencia: vectores, colas dobles y listas.

> Adicionalmente, los vectores de bajo nivel pueden ser considerados como secuencias en el sentido de que todos los algoritmos genéricos de STL que trabajan con secuencias lo hacen también con vectores de bajo nivel. Esto es así gracias a que los punteros funcionan, a todos los efectos, como iteradores de acceso directo.

Además de las características comunes a todas las clases contenedoras, las secuencias poseen:

Constructores por relleno

Reciben como primer parámetro un tamaño y como segundo (omitible) un valor apropiado para construir una secuencia de dicho tamaño con copias del valor. En caso de omitir el segundo parámetro se presupone el constructor por omisión del tipo de los valores.

Constructor por rango

Recibe dos iteradores de entrada apropiados y construye una secuencia que es una copia de los elementos de dicho rango.

assign()

Elimina todos los elementos de una secuencia sustituyéndolos por otros. Hay dos versiones: la primera es análoga al constructor por relleno y la segunda, al constructor por rango.

front()

Devuelve el primer elemento de la secuencia. La secuencia no debe estar vacía.

back()

Devuelve el último elemento de la secuencia. La secuencia no debe estar vacía.

resize()

Modifica el tamaño de la secuencia al valor indicado por su primer parámetro. Ya que puede implicar la creación de nuevos elementos, recibe como segundo parámetro (omitible) un valor apropiado para construirlos. En caso de omitir el segundo parámetro se presupone el constructor por omisión del tipo de los valores.

push_back()

Inserta un elemento al final de la secuencia.

insert()

Hay tres versiones, pero todas reciben como primer parámetro un iterador e insertan en la posición anterior a la indicada por él. La primera inserta un elemento, que recibe como segundo parámetro. La segunda generaliza a la anterior permitiendo la inserción de un cierto número de copias. Por último, la tercera permite insertar los elementos contenidos en un rango que recibe como segundo y tercer parámetros.