Guía de programación C++ STL

Omer Giménez

10 de junio de 2009

Introducción

La STL (Standard Template Library) es una librería estándard de estructuras de datos y algoritmos que forma parte del estándard del C++. Esto garantiza que cualquier compilador de C++ debe incluir de serie esta librería, por lo que cualquier programador puede usarla para sus programas.

El objetivo fundamental de la STL es evitar que los programadores tengan que programarse, una y otra vez, aquellos algoritmos o estructuras de datos fundamentales, como por ejemplo el algoritmo de ordenación, o estructuras de datos como la cola de prioridades (heap) o el diccionario (map).

La STL es una librería compleja, y que internamente hace uso de características bastante avanzadas del C++. Sin embargo, uno puede usar gran parte de la potencia que ofrece la STL de un modo *escandalosamente* fácil. Mi intención al hacer estos apuntes es intentar resumir en unas pocas páginas aquellas partes de la STL que, en mi opinión, pueden resultar de más utilidad para aquellas personas que se estén preparando para participar en concursos de programación, como la OIE (Olimpiada Informática Española) o el ACM-ICPC (International Collegiate Programming Contest).

Por último, sólo me queda indicar que si algún lector detecta errores (de cualquier tipo) en estos apuntes, le rogaría que se pusiera en contacto conmigo para que pudiera corregirlos.

Omer Giménez
omer.gimenez@gmail.com

Capítulo 1

Vectores y matrices: `vector<T>`

Un *vector* (o *tabla*, o *array*) es la estructura de datos más básica: un contenedor que almacena n elementos del mismo tipo en n posiciones consecutivas de memoria. Si el vector se llama `v`, los elementos se denominan `v[0]`, `v[1]`, ..., `v[n-1]`. Para que tu programa use vectores, es necesario que añadas la línea `#include <vector>` al principio de tu código.

¡Cuidado! En un vector `v` de n elementos, `v[n]` no existe, y el elemento `v[0]` existe siempre que el vector no sea vacío, o sea, que $n > 0$. Un programa nunca debe leer (ni, por supuesto, escribir) posiciones no existentes de un vector: en función del sistema operativo y de la posición de memoria que se intente leer, puede pasar que el programa continúe, o interrumpirse debido a un fallo de segmentación.

Avanzado. Un fallo de segmentación es el error que da el sistema operativo cuando detecta que el programa está intentando acceder a memoria que no le corresponde. Sería ideal que siempre que un programa accediera a una posición no reservada, el programa se interrumpiera con un mensaje de error que avisara del error. Desgraciadamente, esto no siempre sucede así en C++, ya que puede ser que accedamos *fuera* del vector pero *dentro* de la memoria que le corresponde al programa. Existen distintas técnicas para detectar este tipo de errores (por ejemplo, Valgrind) pero, con diferencia, lo más útil es intentar no cometer el error desde un principio. Por ello, hay que tener un cuidado exquisito cada vez que escribamos `[ ]` en nuestros programas.

Todos los elementos de un vector son del mismo tipo, y éste se debe especificar en el momento de crearlo: un `vector<T>` es un vector cuyos elementos son de tipo `T`. El tamaño del vector (es decir, el número de elementos de tipo `T` que contiene) también se especifica en el momento de su creación; de otro modo, el vector empieza vacío. Veamos un ejemplo.

```
vector<int> v(3);       //vector de 3 coordenadas enteras
vector<string> vst(10); //vector de 10 strings
vector<bool> w;         //vector de booleanos, vacio
```

A diferencia de los arrays tradicionales del C, los vectores puede cambiar de tamaño durante la ejecución del programa. Más adelante veremos las instrucciones `push_back` y `resize` que nos permiten hacerlo.

Avanzado. Los elementos de un vector se guardan en el heap de memoria libre, y no en la pila del programa: en cierto modo, son equivalentes al uso del `malloc` del C o el `new` del C++, sólo que más sencillos, puesto que no es necesario hacer nunca el `free` o el `delete`. Al modificar el tamaño del vector la STL hace las operaciones necesarias (mover los elementos, reservar nueva memoria, etc.). Como es de suponer, esto tiene un coste, por lo que es preferible evitar las instrucciones que cambian el número de elementos de un vector.

Al crear un `vector<T>` se nos permite pasar un segundo parámetro, de tipo `T`, que indica el valor por defecto con el cual se inicializan los elementos del vector. Si no se especifica ningún valor, como en los ejemplos anteriores, se escoge el valor por defecto del tipo `T`, que es 0 para `int`, `char`, `double`, etc., `false` para `bool`, o `""` (cadena vacía) para `string`.

Para trabajar con un `vector<T>`, al igual que con todas las estructuras de datos que veremos de la STL, se suelen usar los *métodos* que la estructura ofrece. Es fácil encontrar en la web páginas que describen qué métodos usar, y qué hacen estos métodos (podéis consultar, por ejemplo, la documentación oficial del `vector<T>` de la STL, aunque fácilmente encontraréis muchísimos tutoriales, tanto en español como en inglés, que expliquen lo mismo). Por ejemplo, cuando escribimos `v[3]`, en realidad estamos llamando al método `T &vector<T>::operator[](int i)` del `vector<T> v`: un método que recibe un entero `i` y que devuelve (una referencia a) el valor que `v` contiene en la i-ésima posición.

Otros métodos que tiene el vector `v` son `v.size()` (devuelve el tamaño del vector), `v.push_back(e)` (añade un elemento `e` a `v`, incrementando en 1 su tamaño), `v.resize(n)` (modifica el tamaño de `v` para que pase a valer `n`, borrando o añadiendo elementos al final, según convenga).

Los vectores, al igual que todas los contenedores de la STL, tienen la muy útil propiedad de actuar como variables normales y corrientes: podemos asignarlos, compararlos, copiarlos, pasarlos como parámetro, hacer que una función los devuelva, etc. Por ejemplo, podemos escribir código como éste:

```
vector<int> v(10, 3);
vector<int> w = v;    //w es una copia de v
++v[0];               //v y w son ahora distintos
w.resize(0);          //ahora w esta vacio
v = w;                //ahora v tambien esta vacio
bool iguales = v==w; //iguales vale true
```

Avanzado Ésta es una propiedad única que tienen los vectores de la STL (y, en general, todas las estructuras de la STL), y que no comparten ni los arrays tradicionales de C, ni otros tipos de vectores de otros lenguajes, como por ejemplo Java. A esta propiedad se la denomina *value semantics* (semántica de valor).

En concreto, si tenemos dos vectores del mismo tipo `T`, podemos *asignar* uno al otro con el igual `=`, y *compararlos* con el doble igual `==` o el símbolo de distinto `!=`. También podemos compararlos con el `<`, `>`, `>=`, `<=`. Los vectores se comparan según el orden *lexicográfico*, que es el orden con el que se ordenan las palabras del diccionario: se usa el primer elemento del vector para decidir qué vector va antes; en caso de empate, se mira el segundo elemento, y así sucesivamente.

```
vector<char> v(2), w;
v[0] = v[1] = 'a';
w = v;
```

```
  w[1] = 'z';  //v == {'a','a'}, w == {'a','z'}
  cout << "v<w?" << (v<w) << endl;  //true
```

¡Cuidado! Es necesario que el tipo `T` sea comparable; de otro modo, no podrían compararse los vectores. Casi todos los tipos del C++ y la STL son comparables entre sí, por lo que esto no suele ser un problema, a menos que `T` sea una `class` o `struct` definida por nosotros mismos sin un operador de comparación $<$.

Los programadores de C, acostumbrados a trabajar con arrays, suelen olvidarse de la facilidad con la que los vectores pueden ser pasados como parámetros o ser devueltos por funciones, de una forma cómoda y, *generalmente*, eficiente. Por ejemplo:

```
vector<int> lee_vector() {
  int n;
  cin >> n;
  vector<int> v(n);
  for(int i=0;i<n;++i) cin >> v[i];
  return v;
}

void escribe_vector(const vector<int> &v) {
  for(int i=0;i<v.size()-1;++i) {
    cout << v[i] << " ";
  }
  if (v.size()>0) cout << v[v.size()-1];
  cout << endl;
}

void pon_a_cero(vector<int> &v) {
  for(int i=0;i<v.size();++i) v[i]=0;
}
```

¡Cuidado! Los vectores y, en general, todos los tipos de la STL, tienen lo que se llama *semántica de valor* que, además de otras cosas, quiere decir que cuando pasamos un vector como parámetro en una función, lo que se hará es *copiar* todos sus datos a la función que lo recibe. ¡Esto puede ser brutalmente lento! Por lo tanto, siempre que pasamos un vector u otro contenedor de la STL por parámetro es conveniente pasarlo como en el ejemplo anterior: `const vector<T> &` si no vamos a modificar el vector, y `vector<T> &` si nuestra intención es modificarlo. Esto garantiza que, en ambos casos, la función trabajará con el vector original, y no con una copia del mismo. Hay que tener cuidado con este error, ya que puede hacer que un programa que aparentemente funciona para casos pequeños tarde mucho más tiempo del necesario en resolver entradas grandes.

Avanzado. Parecería que las funciones que devuelven vectores también pueden sufrir este problema de copias innecesarias, por ejemplo en `vector<int> v = lee_vector()`. Sin embargo, en casos como éste donde se está creando el contenedor `v`, el compilador detecta que la copia es innecesaria, y hace la correspondiente optimización. Esta optimización no se haría si se hubiera escrito `vector<int> v; v=lee_vector();`.

Otra de las ventajas de los vectores sobre los arrays clásicos es que resultan muy prácticos para almacenar una lista de elementos el tamaño de la cual pueda ir aumentanto. Para ello, se usa el método `push_back` del vector.

```
int main() {
  string palabra;
  vector<string> vs;
  while (cin >> palabra) {
    vs.push_back(palabra);
  }
  cout << "Palabras leidas: " << vs.size() << endl;
}
```

Ésta es una de las pocas excepciones a la *regla* de no cambiar a menudo el tamaño de un vector: el código anterior es razonablemente eficiente.

Avanzado. El motivo es que la STL reserva un espacio de memoria libre después del vector, de modo que éste pueda incrementar su tamaño sin que sea necesario pedir nueva memoria al sistema operativo. Cuando este espacio de memoria, a base de hacer `push_back`, también se acaba agotando y es necesario reservar un nuevo espacio de memoria para almacenar todos los elementos del vector, entonces se reserva el *doble* de memoria de lo que sería estricatamente necesario. De este modo, puede verse que el programa anterior no gasta más del doble de memoria y tiempo para leer los datos del que gastaría de haber conocido desde un principio el número total de elementos a leer.

1.1. Iteradores, recorridos y ordenación

Una de las operaciones más comunes a realizar con un vector consiste en *recorrer* todos los elementos que contiene. Tradicionalmente, esto se programa así:

```
void suma_vector(const vector<int> &v) {
  int suma = 0;
  for(int i=0;i<v.size();++i) {
    suma += v[i];
  }
  return suma;
}
```

El signo < del bucle `for` anterior es debido a que los elementos del vector van del `v[0]` al `v[n-1]`, donde `n` es el tamaño `v.size()` del vector.

¡Cuidado! Técnicamente, `v.size()` no devuelve un número de tipo `int`, sino un número de tipo `size_t` que es, esencialmente, un entero sin signo (un `unsigned int`). Tu compilador puede emitir *warnings* avisándote de que en el `for` anterior estás comparando un `int` con un `unsigned int`. Si quieres no recibir estos avisos, escribe `int n = v.size();` y usa `n` en el `for`, o bien usa `int(v.size())` en vez de `v.size()` dentro del `for`.

Avanzado. No debes tomarte estos *warnings* a broma. Descubre porqué la función `void escribe_vector(const vector<int> &v)` de la sección anterior provoca un fallo de segmentación cuando le pasas un vector vacío.

Algunas pocas veces puede ser necesario recorrer un vector al revés. El código no tiene ninguna complicación, pero lo remarcamos porque es frecuente que uno se olvide de escribir `--i` en lugar de `++i`, o escriba `i>0` en vez de `i>=0`.

```
void escribe_reves(const vector<int> &v) {
  int n = v.size();
  for(int i = n-1;i>=0;--i) {
    cout << v[i] << endl;
  }
}
```

La STL ofrece otro modo de recorrer los vectores distinto al método tradicional de usar un índice `i`. Son los llamados *iteradores*:

```
void suma_vector(const vector<int> &v) {
  for(vector<int>::iterator it = v.begin(); it!=v.end(); ++it) {
    suma += *it;
  }
  return suma;
}
```

Esencialmente, un iterador a un `vector<T>` es una variable de tipo `vector<T>::iterator` que actúa como un *apuntador* a un elemento del vector. Si `it` es un iterador, las operaciones `++it` y `--it` sirven para desplazar el iterador arriba y abajo dentro dle vector, y la operación `*it` devuelve el elemento apuntado por el iterador. Los iteradores `v.begin()` y `v.end()` son dos iteradores especiales que apuntan al primer elemento del vector, y a el primer elemento *fuera* del vector.

Avanzado. Los iteradores no son punteros, pero actúan como si lo fuesen. En cierto modo, un iterador es un puntero *seguro*, que no sólo sirve para apuntar a un valor de un tipo, sinó que además sabe desplazarse por el contenedor donde vive. La idea es que se puedan programar algoritmos en base a *iteradores* que recorren los datos, sin llegar a depender de las *estructuras* que contienen estos datos (este tema no es demasiado relevante para resolver problemas de concurso, de modo que no nos extenderemos más en él).

Los iteradores tienen su razón de ser, y se usan en otras estructuras de la STL; sin embargo, no suele resultar práctico usar iteradores para recorrer los elementos de un vector. Los explicamos porque son necesarios para poder hacer una de las operaciones más importantes en un vector: ordenar sus elementos. La STL permite ordenar muy fácilmente los elementos de un `vector<T>`: basta con hacer una llamada a la función `sort`.

```
  int n;
  vector<int> v;
  while (cin >> n) v.push_back(n);
  sort(v.begin(), v.end());  //ordena un vector de menor a mayor
  for (int i=0;i<v.size();++i) {
    cout << v[i] << endl;
```

```
}
```

Para usar la función `sort`, hay que haber hecho un `#include <algorithm>` al principio del código. Esta función recibe dos parámetros, que son dos iteradores a las posiciones del `vector<T>` a ordenar (un iterador a la primera posición del intervalo a ordenar, y otro a la primera posición fuera del intervalo). Cuando la función retorna, los elementos comprendidos en el intervalo dado aparecen ordenados de menor a mayor.

Avanzado. La función `sort` es eficiente: tarda tiempo proporcional a $n \log n$ en ordenar un vector de n elementos. No sólo eso, sinó que está programada con cuidado para que sea muy rápida en muchas situaciones distintas: excepto para casos bastante concretos, la función `sort` ordenará más rápidamente de lo que podrías conseguir escribiendo tú mismo tu propio algorismo de ordenación. Por lo tanto, úsala.

Al igual que pasa con la comparación de vectores, para ordenar un vector es necesario que los elementos del mismo puedan compararse: los elementos del vector se ordenarán de menor a mayor, en función de lo que signifique su comparación. Por ejemplo, si ordenas `string`, se ordenarán lexicográficamente (como en el diccionario) siguiendo los valores del código ASCII, que dicta que las letras mayúsculas aparecen antes que las letras minúsculas. Si ordenas `pair<T1, T2>`, los elementos se ordenarán primero en función del primer valor de tipo `T1`, y en caso de empate, en función del segundo valor de tipo `T2`. (En la sección de colas de prioridad se dan ejemplos de cómo usar esta propiedad de los `pair`s para ordenar elementos según criterios propios).

Como a menudo puede ser interesante definirse uno mismo el propio criterio de ordenación, la función `sort` permite también ordenar elementos usando una función de comparación que nosotros mismos hayamos definido para la ocasión. Para ello, debemos definir una función que reciba dos elementos `a` y `b` del tipo `T` a ordenar, y que devuelva un `bool`: *cierto*, si el primer elemento `a` es estrictamente más pequeño que `b`, y por lo tanto debería aparecer antes en el vector ordenado; y *falso*, si `a` es equivalente o mayor que `b`. Por ejemplo, para ordenar `string`s en función del número de letras (las palabras más grandes antes) y, en caso de empate, las ordenáramos lexicográficamente, procederíamos del siguiente modo:

```
bool es_menor(const string &a, const string &b) {
  if (a.size()!=b.size()) return a.size()>b.size();
  return a<b;
}

int main() {
  string s;
  vector<string> v;
  while (cin >> s) v.push_back(s);
  sort(v.begin(), v.end(), es_menor);
  for (int i=0;i<v.size();++i) {
    cout << v[i] << endl;
  }
}
```

¡Cuidado! La función de comparación únicamente debe devolver cierto si `a` es menor que `b` (cuando ambos sean iguales o equivalentes, debe devolver falso). Además, la función de comparación debe ser consistente: no puede ocurrir, por ejemplo, que afirme `a<b`, `b<c` y `c<a` a la vez. Una función de comparación errónea puede provocar un fallo de segmentación, ya que llevará a error al algoritmo de ordenación del `sort`.

1.2. Vectores de vectores: matrices

El tipo `T` del `vector<T>` puede ser prácticamente cualquier tipo de datos. En particular, podemos tener vectores de vectores, es decir, vectores donde el tipo `T` es otro vector. Por ejemplo, el tipo `vector<vector<double> >` representa matrices (tablas) de reales.

¡Cuidado! Hay que escribir un espacio entre los símbolos `> >` de `vector<vector<double> >`; de otro modo, el compilador lo confundiría con el símbolo `>>` del `cin`.

Una matriz no es más, pues, que un vector de "filas", donde cada "fila" es un vector de elementos. A diferencia de los arrays bidimensionales tradicionales del C++, no existe la obligación de que todas las filas tengan el mismo número de elementos: cada vector *fila* es independiente de los demás, y la única restricción es que todos ellos deben contener elementos del mismo tipo. Por último, recalcar que los nombres de los tipos involucrados suelen ser tan largos que es una práctica habitual redefinirlos. Veamos un ejemplo.

```
typedef vector<int> VI;
typedef vector<VI> VVI;

int main() {
  VVI M(5);          //matriz de 5 filas vacias (5x0)
  for(int i=0;i<M.size();++i) {
    M[i].resize(3);  //ahora obtenemos matriz (5x3)
  }
  VVI M2(5,VI(3));   //otro modo de tener una matriz 5x3

  //rellenamos las matrices
  for(int i=0;i<M.size();++i) {
    for(int j=0;j<M[i].size();++j) {
      M[i][j]=M2[i][j]=i+j;
    }
  }
}
```

Como puede verse, para crear una matriz de tamaño `n` por `m` debemos escribir `VVI M(n,VI(m))`, donde `VVI` es el tipo matriz y `VI` es el tipo fila. La única cosa que es necesario recordar es que una matriz no es más que un vector de vectores, y que no tiene ningún método adicional a los que ya tuvieran los vectores. Crear una matriz de `n` filas por `m` columnas no es más, pues, que crear un vector de `n` filas, donde cada uno de los elementos de este vector es una fila de `m` columnas. El código `VI(m)` sirve, precisamente, para crear una fila anónima de `m` columnas.

Avanzado. El uso de matrices `vector<vector<T> >` es un poco menos eficiente que el uso de arrays, puesto que cada una de las filas de la matriz se guardan por separado, y se realizan varios accesos de memoria para acceder a una posición (i, j) de la matriz: primero un acceso al tipo matriz (en el stack) para saber dónde guarda sus vectores-fila; un segundo acceso (en el heap) para saber dónde guarda el vector-fila `i`-ésimo sus datos; y un tercer acceso (también en el heap) para acceder al elemento `j`-ésimo de la fila. A cambio, las matrices son más versátiles que los arrays.

Es muy fácil equivocarse cuando se hacen recorridos sobre matrices, y confundir, por ejemplo, el índice que recorre las filas con el que recorre las columnas. Es conveniente usar algún tipo de convenio para equivocarse lo menos posible. Por ejemplo, usar los nombres de variables `i` y `j` para referirse siempre a las filas y las columnas de una matriz. De este modo, sabemos que código como `m[i][j]` tiene buen aspecto, mientras que código como `m[j][i]` es probablemente erróneo. Como ejemplo, mostramos como multiplicar dos matrices.

```
typedef vector<int> Fila;
typedef vector<Fila> Matriz;

//asumimos que las matrices son rectangulares, y que
//el numero de columnas de la primera es igual al
//numero de filas de la segunda
Matriz producto(const Matriz &ma, const Matriz &mb) {
  int nfa = ma.size(), nca = ma[0].size(), ncb = mb[0].size();
  Matriz m(nfa, Fila(ncb));
  for (int i=0;i<nfa;++i) {     //fila de m
    for (int j=0;j<ncb;++j) {   //columna de m
      for (int k=0;k<nca;++k) { //columna en ma, fila en mb
        m[i][j] += ma[i][k]*mb[k][j];
      }
    }
  }
  return m;
}
```

¡Cuidado! En algunos problemas querremos usar matrices para almacenar datos de puntos (x, y) del plano. Seguro que escribiremos cosas como `m[x][y]`, cuando en realidad `x` es un número de *columna*, y `y` es un número de *fila*: estaremos accediendo a la matriz al revés de como hemos hecho en los ejemplos. Podemos elegir el convenio que queramos (no hay ningún problema en ver una matriz como un vector de columnas) pero hay que tener mucho cuidado si mezclamos código que usa el convenio `x`, `y` con código que accede a las matrices con `i`, `j`.

La misma idea de las matrices bidimensionales puede usarse para tener matrices tridimensionales o de más dimensiones. Sin embargo, la notación excesivamente engorrosa (es necesario usar varios typedefs, y es molesto inicializar una matriz de un tamaño concreto) y los niveles adicionales de indirección (vectores de vectores de vectores etc.) hacen que ésta no sea la mejor opción.

1.3. Cuándo es preferible usar arrays a vectores

Existen unas pocas situaciones en las que es preferible usar arrays tradicionales en vez del tipo `vector<T>`. (En este documento no explicaremos cómo usarlos; puesto que éste era el modo usual de trabajar antes de tener los vectores de la STL, la gran mayoría de los manuales de C o de C++ explican los arrays). Ya que hemos comentado ampliamente las ventajas de los vectores, sería justo indicar cuándo es preferible usar arrays..

Notación Para declarar vectores de 3 dimensiones (o más) cuyo tamaño se conozca con antelación (es decir, que no dependa de la entrada), como a veces ocurre en algunos problemas de programación dinámica, suele ser más cómodo trabajar con arrays de C que con vectores.

Eficiencia (1). En general, acceder a un elemento de un vector es igual de eficiente que acceder a un elemento de un array a través de un puntero (por ejemplo, cuando se declara un array con `malloc` o `new`): el compilador optimiza la llamada a la función `operator[]` para que esto sea así. Hay algunos casos, sin embargo, donde el compilador puede hacer optimizaciones adicionales con los arrays. Los arrays también serán más veloces si se usan para matrices de varias dimensiones, o si se compila sin optimizaciones.

Eficiencia (2). El código que inicializa o cambía el tamaño de un vector se asegura de limpiar o inicializar todos los elementos del mismo. En algunas ocasiones concretas, esta operación puede ser demasiado lenta; usar memoria dinámica o arrays estáticos puede resultar más eficiente.

Comodidad. Se puede poblar los elementos de un array en la misma línea dónde se declara, cosa que no es posible en C++ en la actualidad (lo será con el nuevo estándard C++0x que se está preparamdo). El ejemplo más típico es el siguiente:

```
void vecinos(int x, int y) {
  int dx[] = {0, 1, 0, -1};
  int dy[] = {1, 0, -1, 0};
  string nombres[] = {"sur", "este", "norte", "oeste"};
  for (int i=0; i<4; ++i) {
    cout << nombres[i] << ": (" << x+dx[i] << "," << y+dy[i] << ")" << endl;
  }
}
```

Capítulo 2

Pilas y colas: `stack<T>` y `queue<T>`

El modo en que se guardan los datos en memoria está directamente relacionado con la forma en la que se espera *acceder* a ellos. Los vectores, por ejemplo, ofrecen uno de los tipos de acceso más útiles que existen: acceso *aleatorio*, es decir, el programa puede acceder a el dato `v[i]` del vector que desee, en cualquier momento, teniendo únicamente que indicar el índice `i` del mismo. A cambio, las operaciones de añadir o eliminar elementos del vector pueden resultar costosas.

En aquellos casos concretos en los que no se necesita acceso aleatorio a los elementos, puede ser preferible utilizar alguno de los otros contenedores de la STL. Dos de los más sencillos son la pila (*stack*) y la cola (*queue*). El modo más visual de verlos es decir que un *stack* es como una pila de platos (el último plato que apilamos es el único plato que podemos sacar de la pila) y que un *queue* es como una cola del supermercado (la persona que lleva más tiempo en la cola es la persona a quien debemos atender).

Si nuestro programa necesita trabajar con datos de un modo que se asemeja a una pila o una cola, es preferible usar estos contenedores, en vez de algún otro. Por ejemplo, usando pilas y colas nunca debemos preocuparnos del tamaño (el `.size()`) del contenedor, cosa que no pasaría si, por ejemplo, usáramos vectores para guardar los elementos.

Al igual que los vectores (y que todos los contenedores de la STL) los tipos que almacenan estos contenedores son variables: con `stack<T>` declaramos una pila de tipo `T`, mientras que con `queue<T>` declaramos una cola de tipo `T`. Para usar estos contenedores es necesario hacer `#include <stack>` o `#include <queue>`.

Ambos tipos tienen, al igual que un vector, los métodos `size()` (número de elementos almacenados), `empty()` (retorna cierto si vacío). Los métodos más característicos son el método `push(const T &t)`, que añade una copia del elemento `t` a la pila o cola, y `pop()`, que borra un elemento. Los elementos se añaden o se borran según las reglas de la pila o la cola: por ejemplo, hacer un `push` seguido de un `pop` es equivalente a no hacer nada en una pila, pero supondrá eliminar el elemento más antiguo de una cola y poner, al final de la misma, un nuevo elemento. Ninguno de estos métodos, sin embargo, sirve para *consultar* qué hay dentro del contenedor; para ello hay que usar el método `T &front()` (para colas) y `T &top()` (para pilas) que retorna el elemento que sería el siguiente en salir.

¡Cuidado! Aunque para acceder a pilas y colas no haya que usar índices como con los vectores, sigue siendo todavía posible cometer fallos de segmentación, haciendo un `pop()`, un `front()` o un `top()` con una cola o una pila vacía.

A continuación, mostramos cómo usar un `stack<string>` para leer una secuencia de palabras de la entrada y escribirla del revés.

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
  stack<string> pila;
  string s;

  while (cin >> s) pila.push(s);
  while (not pila.empty()) {
    cout << pila.top();
    pila.pop();
  }
}
```

En el siguiente ejemplo usamos dos colas para emparejar palabras. Suponiendo que recibimos una secuencia de palabras, queremos emparejar cada palabra que empieza con mayúscula con una palabra que empiece por minúscula, cumpliendo el orden de entrada: la primera palabra con mayúscula deberá emparejarse con la primera palabra con minúscula, y así sucesivamente. Al final, devolvemos el número de palabras que no ha sido posible emparejar.

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
  queue<string> qmay, qmin;
  string s;
  while (cin >> s) {
    if (s[0]>='A' and s[0]<='Z') qmay.push(s);
    else qmin.push(s);
  }

  while (not qmay.empty() and not qmin.empty()) {
    cout << qmay.front() << "-" << qmin.front() << endl;
    qmay.pop();
    qmin.pop();
  }

  cout << "Han sobrado " << (qmay.size()+qmin.size())
       << " palabras." << endl;
}
```

2.1. Sobre los usos de las pilas y las colas

Pilas y colas son contenedores que sirven habitualmente para modelar el *trabajo pendiente*, por decirlo de algún modo. Una cola modela la idea de "el trabajo que lleva más tiempo pendiente es el

primero que deberíamos tratar", mientras que una pila indica que "el último trabajo que recibimos es el que deberíamos resolver primero". Según qué estemos programando querremos usar una u otra.

Por ejemplo, cuando nuestro programa consiste en varios procedimientos que se llaman uno a otro, estamos creando *implícitamente* una pila de llamadas pendientes: el `main` llama a un procedimiento; éste, durante su ejecución, llama a otro; etc. Hasta que no finaliza el último procedimiento en ser llamado, no puede continuar la ejecución de aquél que le llamó. Se trata, en efecto, de una *pila* de llamadas.

Avanzado. La parte de memoria donde se declaran las variables locales de un procedimiento se llama *stack* (pila) porque éste es el papel que hace cuando un procedimiento llama a otro: sirve para almacenar, en forma de pila, las variables locales y los parámetros que tienen los procedimientos. Por este motivo siempre es posible simular un programas recursivo usando pilas; equivalentemente, también es el motivo por el que los programadores que saben hacer programas recursivos son capaces de evitar a veces el uso de `stack<T>`.

Otro uso clásico de las pilas es para procesar expresiones parentizadas o cálculos matemáticos: cada vez que abrimos un nivel de paréntesis en una expresión, se nos pide que *olvidemos* provisionalmente lo que hubiera fuera de los mismos, que operemos dentro de los paréntesis, y que utilicemos el resultado con aquello que hubiera justo en el entorno de los paréntesis. Se trata, en efecto, de realizar una pila de cálculos.

En cuanto a las colas, podemos decir que, a diferencia de los ejemplos anteriores, se usan cuando quereremos almacenar trabajo pendiente que no debe realizarse ahora, sinó más tarde. Los sistemas operativos, por ejemplo, permiten que se ejecuten varios programas en paralelo gracias a que tienen una *cola de procesos* que usan para repartir la CPU entre los distintos programas en ejecución: el programa que lleva más tiempo esperando es aquél a quien le dejarán usar la CPU un determinado instante de tiempo, para volver luego al final de la cola. (Aunque este es un ejemplo ilustrativo, la realidad es bastante más compleja, puesto que algunos programas tienen mayor prioridad que otros).

Uno de los ejemplos más característicos de colas son los recorridos en anchura. Pongamos, por ejemplo, que queramos ordenar todas las personas del mundo en función de la "distancia-en-amigos" que tienen conmigo: si somos amigos, nuestra distancia es uno; si no lo somos, pero tenemos un amigo en común, nuestra distancia es dos; y así sucesivamente. Un modo de proceder sería empezar formando una cola con todos nuestros amigos directos. Uno por uno, los vamos quitamos y añadimos a la cola todos sus amigos que no sean amigos nuestros. Veríamos, pues, como se acumularían en esta cola personas de distancia 2. Al sacar estas personas de la cola y poner en ella sus amigos, obtendríamos nuestros amigos de distancia 3, y así sucesivamente. Aunque el ejemplo dado pueda parecer un poco banal, la misma técnica se puede dar para resolver otros problemas.

Capítulo 3

Colas de prioridad: priority_queue<T>

Una *cola de prioridad* es una estructura donde se guardan datos de modo que el *mayor* de ellos pueda ser consultado o eliminado. En cierto modo, es como una *cola* o una *pila* donde los elementos no se ordenan según el orden en el que se han insertado, sinó según su valor.

La cola de prioridad que la STL ofrece se llama `priority_queue<T>`, donde `T` es el tipo de datos que se guardarán en ella. Como los elementos tienen que estar ordenados, es necesario que los elementos del tipo `T` sean comparables entre sí usando el operador `<` (o alguna función de comparación apropiada). Para usar una `priority_queue` hay que incluir la cabecera `<queue>`.

Avanzado. La `priority_queue<T>` de la STL se implementa mediante un *heap* sobre un `vector<T>`. Por lo tanto, insertar o eliminar un elemento en una cola de prioridad de n elementos tarda tiempo proporcional a $\log n$, y consultar el elemento mayor tarda tiempo constante. Es posible cambiar el tipo del contenedor `vector<T>` sobre el que se monta el *heap*, algo que nunca (o casi nunca) es necesario.

3.1. Operaciones básicas

Una `priority_queue<T> pq` tiene 3 operaciones básicas, esencialmente las mismas que un `stack` o una `queue`. Para añadir un elemento `e` se usa el método `pq.push(e)`; para recuperar el elemento mayor entre los existentes se llama a `pq.top()`, y para eliminarlo se llama a `pq.pop()`. Además, también ofrece los métodos típicos de otros contenedores, como `pq.empty()` para saber si esta vacía, `pq.size()` para conocer el número de elementos, y `pq.clear()` para vaciarla.

¡Cuidado! Debe seguirse una precaución similar que con las pilas y con las colas: no puede llamarse al método `top()` ni al `pop()` si la `priority_queue` está vacía.

Una curiosidad: la `priority_queue<T>` no es ni una cola ni una pila, pero la implementación de la STL nos hace incluir el fichero `<queue>` para poder usarla (como si fuera una cola) y llamar al método `top()`, y no al `front()`, para acceder al elemento mayor (como si fuera una pila).

Mostramos ahora un ejemplo de como usar una `priority_queue<string>` para leer varios `string` de la entrada y escribirlos en orden, de mayor a menor (el mismo efecto se hubiera podido conseguir ordenando un `vector<string>` y recorriendo el `vector` al revés).

```
priority_queue<string> pq;
string s;
while (cin >> s) pq.push(s);
while (not pq.empty()) {
  cout << pq.top() << endl;
  pq.pop();
}
```

El ejemplo anterior no es muy interesante porque primero se insertan todos los elementos, y luego se retiran. La utilidad de una `priority_queue` viene cuando es necesario intercalar las operaciones de *insertar* (cualquier elemento) y *consultar* o *eliminar* (el mayor). Esta situación puede ocurrir, por ejemplo, cuando se intenta simular algún proceso, como una cola de trabajos pendientes: en la cola van entrando trabajos de todo tipo, y se van resolviendo aquellos que son más prioritarios.

Avanzado. La implementación más eficiente del algorismo de Dijkstra para encontrar los caminos mínimos en un grafo también utiliza una `priority_queue`.

3.1.1. Ordenación con `pair`

A menudo se quiere distinguir entre el *elemento* a guardar en la cola de prioridades, y la *prioridad* que le asignamos en la cola. Sin embargo, ya hemos visto que la `priority_queue` no permite hacer esta distinción: la prioridad del elemento es su propio valor, en comparación con los demás elementos de la cola.

Existen varios modos de resolver este problema. El más sencillo e intuitivo consiste en guardar en la cola *pares* de datos formados por el elemento y su prioridad, y usando un criterio de ordenación que compare estos pares en función de su prioridad. Para ello podemos usar el tipo `pair<T1,T2>` de la STL, que a partir de dos tipos `T1` y `T2` cualesquiera crea un nuevo tipo que contiene dos variables, una de cada tipo. Si `p` es un `pair<T1,T2>`, se tiene que `p.first` es una variable de tipo `T1` y `p.second` es una variable de tipo `T2`. Los `pair<T1,T2>` se comparan entre sí usando la variable de tipo `T1` y, en caso de empate, usando la de tipo `T2`. Para usar el tipo `pair` es necesario hacer un `#include <algorithm>`.

Resolvemos a continuación el problema siguiente. Se recibe por la entrada una secuencia de trabajos, donde cada trabajo contiene una descripción (`string`) y una prioridad (`int`). Una entrada con descripción `"LIBRE"` y prioridad `0` es un valor especial: indica que se dispone de tiempo libre para procesar uno de los trabajos pendientes de prioridad máxima que todavía no haya sido procesado (cualquiera de ellos, por ahora). El trabajo procesado debe escribirse por la salida y dejar de contabilizarse.

```
typedef pair<int, string> PIS;

int main() {
  PIS p;
  priority_queue<PIS> pq;
  //leemos trabajo + prioridad
  while (cin >> p.second >> p.first) {
    if (p.first==0 and p.second=="LIBRE") {
      if (not pq.empty()) {
        p = pq.top();
        pq.pop();

        cout << p.second << endl;
```

```
      }
    }
    else pq.push(p);
  }
}
```

¡Cuidado! Siempre que se trabaja con `pair` es conveniente usar los `typedef` para evitar escribir el nombre entero del tipo una y otra vez. En general, un código más corto, siempre que no sea críptico, es más fácil de leer. Desgraciadamente, los nombres `first` y `second` del `pair` son bastante largos y poco indicativos: esto es una potencial fuente de errores, y es conveniente prestar mucha atención cuando se trabaja con `pair`.

Avanzado. Hay quien usa código como `#define x first` y `#define y second` para no tener que escribir nombres tan largos (u otros nombres más indicativos en vez de `x` e `y`). No creo que este tipo de "renombramientos" con `#define` sean aconsejables en programas largos, pero si uno sabe lo que hace, tal vez no sea mala idea al hacer programas cortos, como los de un concurso de programación.

El ejemplo anterior procesa los trabajos en función de su prioridad. Sin embargo, cuando se encuentra con dos trabajos de idéntica prioridad en la cola, usará el criterio de la segunda variable del `pair` para desempatar, por lo que siempre escogerá aquél trabajo cuya descripción sea *mayor* según el criterio de comparación de los `string`: es decir, la descripción que aparecería en el último lugar si las ordenásemos siguiendo el criterio del diccionario. A continuación modificamos el código anterior para que, en caso de empate de prioridades, devuelva el trabajo que entró *primero* en la cola. Como puede verse, damos una vuelta de tuerca más a la misma idea de antes.

```
typedef pair<int, int> PII; //prioridad, orden (negativo)
typedef pair<PII, string> PPIIS;

int main() {
  int t = 0, prio;
  string desc;
  priority_queue<PPIIS> pq;

  while (cin >> desc >> prio) {
    if (prio==0 and desc=="LIBRE") {
      if (not pq.empty()) {
        PPIIS p = pq.top();
        pq.pop();

        cout << p.second << endl;
      }
    }
    else {
      ++t;
      pq.push(PPIIS(PII(prio, -t), desc);
    }
  }
```

```
}
```

Usamos `-t` para hacer que los *primeros* trabajos en aparecer tengan una prioridad *mayor* ($-1 > -2 > -3 > \cdots$), por lo que saldrán *antes* de la `priority_queue`. El truco del cambio de signo es un modo rápido de conseguir que los elementos se extraigan de *menor* a *mayor* (por lo que respecta al tiempo `t`, puesto que la prioridad `prio` se sigue extrayendo de mayor a menor). En el código anterior podemos ver también que, si queremos crear un *valor* `(v1,v2)` de tipo `pair<T1,T2>` para insertar en la `priority_queue`, podemos escribir `pair<T1,T2>(v1,v2)`. En el código, el valor `PII(prioridad,-t)` sirve únicamente para inicializar la primera variable del `PPIIS`, y éste, a su vez, sirve para ser insertado en la `priority_queue`.

3.2. Ordenación redefiniendo el `operator<`

La STL ofrece la posibilidad de cambiar el criterio de ordenación de los elementos guardados en una `priority_queue`, sin necesidad de tener que recurrir a trucos como el del `pair` descrito anteriormente. Describimos a continuación el primero de dos modos distintos de hacer el cambio el criterio.

Cada `priority_queue<T>` guarda internamente un criterio de comparación para ordenar sus elementos. Por defecto, este criterio es `less<T>`, que consiste en usar el operador `<` para comparar los elementos.

Avanzado. Este criterio `less<T>` es un *functor* (una clase que puede ser llamada como si fuera una función, porque ofrece un método `operator()` público) templatizado según `T` que no hace más que llamar al `operator<` del tipo `T`.

La gran mayoría de los tipos, tanto los básicos (`int`, `char`, `double`) como los de la STL (`string`, `vector`, `map`, `pair`), tienen ya definido un criterio de ordenación. En cambio, cuando se crea un tipo nuevo (con `struct` o `class`), éste no recibe ningún criterio de ordenación por defecto. Por lo tanto, si queremos una `priority_queue` donde se guarde un tipo de datos definido por nosotros, podemos también especificar qué criterio de ordenación deberá seguir, definiendo el `operator<` de nuestro tipo.

¡Cuidado! De hecho, estamos en la obligación de definirlo: de lo contrario se generaría un error de compilación *dentro* del código de la propia `priority_queue` (en concreto, dentro del `less<T>`), ya que el compilador no sabría como comparar los elementos con el $<$. El mensaje de error de este tipo de errores puede llegar a ser bastante críptico.

Vemos ahora como solucionar el problema anterior de los trabajos con prioridades sin necesitar de usar `pair`.

```
struct Trabajo {
  int prio, t;
  string desc;
};

//cierto si 'b' tiene preferenia respecto 'a'
bool operator<(const Trabajo &a, const Trabajo &b) {
  if (a.prio!=b.prio) return a.prio<b.prio;
  return a.t>b.t;
}

int main() {
```

```
  int t=0;
  Trabajo tra;
  priority_queue<Trabajo> pq;

  while (cin >> tra.desc >> tra.prio) {
    if (tra.prio==0 and tra.desc=="LIBRE") {
      if (not pq.empty()) {
        Trabajo p = pq.top();
        pq.pop();

        cout << p.desc << endl;
      }
    }
    else {
      ++t;
      tra.t=t;
      pq.push(tra);
    }
  }
}
```

Como puede verse, la única dificultat adicional consiste en declarar una función de comparación `operator<` que reciba dos variables de tipo `const T &` y las compare según el criterio que queramos. El programa es un poco más largo que cuando se usaba el `pair`, y con nuestro `struct Trabajo` no podemos usar la notación `pair<T1,T2>(v1,v2)` para crear un valor `(v1,v2)` como habíamos hecho antes, pero a cambio se gana claridad de código: es más difícil cometer errores escribiendo este programa que escribiendo el anterior.

¡Cuidado! La función de comparación no puede ser cualquier función: tiene que representar una ordenación válida de los elementos. La situación es idéntica a la que ocurre cuando se intenta crear una función de comparación para el `sort` de vectores. Por ejemplo, no puede ser que ocurra `(x<y)` y `(y<x)` a la vez, o que `(x<y)` y `(y<z)` pero que en cambio `(x<z)` devuelva falso, o que el resultado de la comparación cambie durante la ejecución del programa. Si se escribe una función de comparación errónea la `priority_queue` no funcionará correctamente, pudiendo colgarse o dar lugar a un error de segmentación. Se permite, en cambio, que hayan elementos `x,y`, distintos entre sí, pero tales que ninguna de las comparaciones `(x<y)` o `(y<x)` devuelva cierto: esto se interpreta como elementos *distintos* pero con prioridades *equivalentes*.

Avanzado. El C++ permite dotar a nuestros tipos, como el `struct Trabajo`, de la facultad de crear valores temporales usando una notación como `Trabajo(desc, prio, t)`. Con ello, es posible prescindir del uso de `pair`: el código resulta más largo de escribir, pero a cambio el programa resulta más claro. Para conseguirlo, es necesario definir un constructor de inicialización `Trabajo(string d, int p, int t)` y un constructor por defecto `Trabajo()`. Los dos constructores son necesarios: el estándard del C++ especifica que, al definir el constructor de inicialización, deja de generarse automáticamente el constructor por defecto, constructor que necesita el `vector<Trabajo>` sobre el que se monta la `priority_queue`. Por lo tanto, el codigo completo del `struct Trabajo` debería ser el siguiente.

```
struct Trabajo {
```

```
  int prio, t;
  string desc;
  Trabajo(int p, int t, string d): prio(p), t(t), desc(d) {}
  Trabajo() {}
};
bool operator<(const Trabajo &a, const Trabajo &b) {
  if (a.prio!=b.prio) return a.prio<b.prio;
  else return a.t>b.t;
}
```

3.3. Ordenación cambiando el `less<T>`

Por último, vemos un último modo de cambiar el criterio de ordenación que no requiere redefinir ningún operador `<`. Esto es interesante, por ejemplo, cuando queremos tener una `priority_queue<T>` de algún tipo `T` que ya tiene un `operator<` predefinido, o cuando hay la necesidad de tener varias `priority_queue<T>`, cada una de ellas con un criterio de ordenación distinto. Para ello, se tiene que *extender* la definición de una `priority_queue<T>` a `priority_queue<T, vector<T>, compara>`, donde `compara` es el criterio de ordenación que usará nuestra `priority_queue`.

Avanzado. El `vector<T>` es el contenedor donde se forma la `priority_queue`; es necesario escribirlo para que la `priority_queue` pueda recibir `compara` como tercer parámetro del *template*.

A diferencia de la función `sort`, para pasar el criterio de comparación a una `priority_queue` es necesario crear un `struct` que contiene una función de comparación `operator()`. Esto hace que sea un poco más engorroso de declarar de lo necesario. Mostramos a continuación cómo tener dos tipos de colas de prioridades de tipo `string` que usan criterios diferentes.

```
//orden lexicografico al reves: asi, las que aparezcan
//antes en el diccionario son mas prioritarias.
struct compara_diccionario{
  bool operator()(const string &a, const string &b) {
    return a>b;
  }
};

//las mas largas tienen mayor prioridad; en caso de empate,
//tiene prioridad la primera en el diccionario.
struct compara_size{
  bool operator()(const string &a, const string &b) {
    if (a.size()!=b.size()) return a.size()<b.size();
    return a>b;
  }
};

typedef priority_queue<string, vector<string>,
                       compara_diccionario> PQ_dicc;

typedef priority_queue<string, vector<string>,
                       compara_size> PQ_size;
```

Capítulo 4

Conjuntos y diccionarios: `set<K>` y `map<K,T>`

Un *diccionario* es una estructura donde podemos guardar *datos* ordenados según una *clave*, de tal modo que resulte muy eficiente buscar el dato conociendo su clave correspondiente. Por ejemplo, en un diccionario real, la clave es una palabra, mientras que el dato guardado es la definición de la misma.

La STL ofrece diversas estructuras de datos que pueden ejercer funciones parecidas a las de un diccionario. La más importante es el `map<K, T>`, donde `K` es el tipo de la clave y `T` es el tipo de los datos. Se requiere, además, que los elementos de tipo `K` sean ordenables entre sí usando el operador `<` (o alguna función de comparación apropiada). El `set<K>` es una versión simplificada del `map`, donde sólo guardamos claves, sin datos asociados. Es, pues, un *conjunto* en vez de un diccionario. Veremos primero el `map`, y luego comentaremos sus diferencias con el `set`. Los ficheros de cabecera que es necesario incluir, como cabría esperar, son `<map>` y `<set>`.

Avanzado. Los otros tipos de datos de la STL que pueden hacer de diccionario son el `multi_map` (un diccionario donde es posible guardar varios datos distintos con la misma clave) y el `hash_map` (un diccionario implementado de un modo distinto al `map`). Generalmente, no es necesario usar ninguno de estos tipos para resolver problemas de concurso de programación.

La principal característica que se espera de un diccionario es que las operaciones de *insertar* un nuevo dato con su clave, *buscar* si una clave existe, y *recuperar* el dato asociado, sean veloces. En concreto, se espera que tarden tiempo proporcional al logaritmo del número de elementos almacenados en el diccionario. Esto se corresponde, intuitivamente, con el tiempo que tardaríamos a encontrar una palabra en un diccionario real siguiendo el algoritmo de búsqueda dicotómica: abrimos el diccionario por la mitad, vemos en qué mitad del diccionario está la palabra buscada, volvemos a abrir por la mitad de la parte donde está la palabra, etc. Como a cada paso descartamos la mitad del trozo de diccionario que nos interesaba, tardamos tantos pasos como veces podamos dividir por la mitad el número de palabras del diccionario, es decir, el logaritmo en base 2 de este número.

Avanzado. La implementación usual de los tipos `map` y `set`, al igual que el `multi_map` y `multi_set`, es usando árboles binarios de búsqueda, con balanceo *red-black*. Los tipos `hash_map` y `hash_set` se implementan, como su nombre indica, con *hash tables* (tablas de dispersión). En general, se gana un poco de eficiencia usando estos últimos tipos, pero a cambio se pierde el acceso secuencial ordenado que permiten los arboles binarios.

4.1. Operaciones básicas

En un `map`, podemos insertar y recuperar datos de modo muy parecido a como usaríamos un `vector`: escribiendo `m[c]=d;` *insertamos* un dato `d` con clave `c` (o *modificamos* el dato antiguo, si la clave `c` ya estuviera en el diccionario), y escribiendo `m[c]` *recuperamos* el dato con clave `c`.

```
map<int, string> jug;
jug[1] = "Johnny";
jug[23] = "Jordan";
cout << "1: " << jug[1] << endl;
if (jug[23]=="Jordan") {
    cout << "y ademas, tenemos a Jordan!" << endl;
}
```

Por ejemplo, un `map<int, T>` funciona de un modo parecido a un `vector<T>`, con la ventaja de que no necesitamos preocuparnos de su tamaño, o de si accededemos fuera de rango, porque cualquier entero puede actuar como clave. En el ejemplo anterior, hubiéramos podido escribir también `jug[-1]="Nadie";`, sin ningún problema.

Escribir `m[c]` siempre sirve para *recuperar* un dato con clave `c`. ¿Qué ocurre cuando no hay ningún dato con clave `c`? En otros lenguajes de programación, esto provocaría un error en tiempo de ejecución. La STL, sin embargo, hace lo siguiente: toma las medidas necesarias para que este intento de recuperación no provoque ningún error. Para ello, el `map m` *inserta* un dato con clave `c` y dato vacío (0 si el dato es de tipo `int`, `double` o `bool`, `""` si el dato es de tipo `string`, etc.), y retorna ese dato vacío. Por lo tanto: *nunca* hay que escribir `m[c]` a menos que estemos seguros de que exista la clave `c`, o que queramos insertar en el `map` la clave no existente. A continuación, un ejemplo donde se muestra un código *erróneo* para descubrir la primera entrada con clave positiva de un `map`.

```
//codigo erroneo para encontrar la primera entrada con clave
//positiva de un map
int busca_positiva(map<int, string> &m) {
  int c = 1;
  while (m[c]=="") ++c; // (*)
  return c;
}
```

Este código es erróneo por dos motivos. Primero, porque no estaría considerando aquellas entradas que tuvieran dato `""`, ya que las confundiría con claves no existentes. Segundo, y más importante, la línea marcada con un `(*)` busca entradas, pero cada vez que no encuentra una clave, *inserta* una nueva entrada vacía en el `map`. Por ejemplo, si empezáramos con un `map` de un solo elemento con clave `100000`, este código insertaría 99999 claves con dato vacío antes de encontrarlo. Si hubiéramos escrito `const map<int,string> &m` como parámetro en la función, el compilador no nos hubiera dejado compilar debido a esta línea, puesto que estaríamos inclumpiendo la condición de `const`.

A continuación, un ejemplo donde, gracias a esta propiedad de insertar elementos vacíos, podemos contar el número de veces que aparece cada palabra por la entrada usando una única línea de código.

```
map<string, int> m;
string s;
while (cin >> s) ++m[s]; // (*)
```

En la línea `(*)`, si `s` aparece en el diccionario incrementamos su valor, y si `s` es nueva entonces se inserta el dato vacío `0`, y a continuación se incrementa, por lo que pasa a `1`: en efecto, cuando acaba el `while` hemos creado un diccionario donde a cada palabra que apareciera por la entrada se le asocia el número de apariciones.

Para saber si una clave está o no en el `map` sin insertarla debemos usar el método `find`: este método nos devuelve un *iterador* que nos indica si ha encontrado o no la clave, y qué dato asociado tiene la misma. Más adelante estudiaremos los iteradores de un `map`; por ahora, lo único que necesitamos saber es que para saber si la clave existe o no existe en el `map m` sólo necesitamos hacer `m.find(c)` y comparar el valor que retorna con `m.end()`: si son iguales, es que la clave *no* existe; de otro modo, sabemos que escribir `m[c]` es seguro, y que devolverá el dato asociado.

Avanzado. Haciendo lo descrito anteriormente estaríamos realizando dos búsquedas, una primera `m.find(c)` para saber si el dato existe, y una segunda `m[c]` para encontrar el dato asociado. En realidad, el iterador que devuelve el `find` ya contiene el dato asociado, sin que necesitemos buscarlo de nuevo en el diccionario: habría que guardar la respuesta en una variable de tipo iterador, compararlo con `m.end()`, y si son distintos, usar este mismo iterador para acceder al dato. Más adelante veremos cómo se hace.

Por ejemplo, presentamos ahora un código algo más correcto que el anterior para buscar la primera entrada con clave positiva de un `map`:

```
//codigo (lento) para encontrar la primera entrada con clave positiva
//de un map, asumiendo que exista alguna
int busca_positiva(const map<int, string> &m) {
  int c = 1;
  while (m.find(c)==m.end()) ++c;
  return c;
}
```

Este código usa el `find` para saber si una clave existe o no sin insertarla. Sin embargo, todavía tiene dos defectos: primero, que si no hubiera ninguna entrada con clave positiva en el `map`, el código se quedaría buclado en el `while`; segundo, que es extremadamente lento, puesto que prueba todas las claves una por una hasta encontrar la más pequeña. Luego veremos cómo resolver ambos problemas.

Para eliminar el elemento con clave `c` de un `map` basta con hacer `m.erase(c)`. El `erase` también funciona si recibe un iterador (como los devueltos por el `find`) en vez de una clave. Al igual que con otras estructuras de la STL, también podemos hacer `m.size()` para conocer el número de elementos almacenados, `m.empty()` para saber si está o no está vacío, y `m.clear()` para borrar todos los elementos del `map`.

4.2. Recorridos

Una de las operaciones más usuales de un `map` es la de *recorrer* todos los elementos que contiene. Internamente, el `map` guarda todos los elementos que hemos insertado ordenados según su clave: podemos recorrer los datos guardados siguiendo este orden, usando *iteradores*. Un iterador es, en cierto modo, la *posición* que ocupa un par clave-dato dentro del diccionario. A través de un iterador podemos acceder no sólo a la clave y al dato, sino también al elemento siguiente y anterior del diccionario. De este modo podemos recorrerlo, elemento a elemento, de un modo *idéntico* al que ya vimos con los vectores.

Avanzado. Un *iterador* es, en realidad, poco más que una clase que hace de *wrapper* a un *puntero*, al que se le han definido unas cuantas operaciones como incrementar, decrementar, comparar, etc. Lo verdaderamente útil de los iteradores no es tanto lo que pueden hacer por ellos mismos, sinó el hecho de que otros contenedores de la STL ofrecen también sus propios iteradores siguiendo una sintaxis común.

Si `m` es un `map<K,T>` y `it` es un iterador que apunta a un elemento del diccionario, se tiene que `it->first` es la *clave* del elemento, de tipo `const K`, y que `it->second` es el *dato* correspondiente, de tipo `T`. (La notación `->first` y `->second` se debe a que un iterador actúa como un puntero a un `pair<const K,T>`). Dado un iterador `it`, las operaciones `++it;` y `--it;` hacen que el iterador apunte al siguiente o anterior elemento del diccionario. El método `m.begin()` devuelve un iterador que apunta al primer elemento del `map`, mientras que `m.end()` devuelve un iterador que apunta *una posición después* del último elemento del diccionario. (Este era el iterador que devolvía el `m.find(c)` para indicar que no había encontrado la clave `c`.) Para recorrer un `map`, por lo tanto, basta con empezar con el iterador `m.begin()` e irlo incrementando con `++it;` hasta comprobar que es igual a `m.end()`, momento en el que habremos acabado de recorrer por orden todos los elementos del `map`.

¡Cuidado! Al igual que cuando se trabaja con vectores, hay que prestar atención a todos los posibles errores. Es ilegal incrementar un iterador que sea igual al `m.end()`, o decrementar un iterador igual al `m.begin()`. No se puede acceder ni al `it->first` ni al `it->second` de un iterador que sea igual al `m.end()`. No es posible modificar la clave `it->first`, ya que de hacerlo el `map` podría dejar de estar ordenado (por ese motivo se tiene que `it->first` es de tipo `const K`, para que el compilador detecte este error). Por último, en un `map` vacío se cumple que `m.begin()==m.end()`.

Por último, sólo nos falta decir cómo crear una variable de tipo iterador. No existe un tipo `iterator`, a secas, ni tan siquiera `map_iterator`, porque cada tipo de `map<K,T>` tiene sus propios iteradores; el tipo de un iterador a un `map<K,T>` se llama `map<K,T>::iterator`. Con esto, ya podemos recorrer todos los elementos de un `map`. Por ejemplo, completamos el programa que calculaba cuántas veces aparecía cada palabra por la entrada, haciendo que escriba el número de apariciones *ordenadamente* (según el orden lexicográfico de las palabras que aparecen, que hacen de claves en este ejemplo).

```
map<string, int> m;
string s;
while (cin >> s) ++m[s];

map<string, int>::iterator it;
for (it = m.begin(); it != m.end(); ++it) {
  cout << it->first << ": "
       << it->second << " veces" << endl;
}
```

¡Cuidado! Los iteradores *no* deben compararse entre sí con los operadores $<$ y $>$. Hay que limitarse a usar los operadores `!=` o `==`: por este motivo hemos escrito `it!=m.end()` y no `it<m.end()`. El motivo es que la comparación entre iteradores se hace según los valores numéricos de sus punteros y no según el orden en el que aparecen en el `map`.

Como es un poco engorro tener que volver a escribir el tipo exacto del `map` cada vez que se quiere declarar un iterador, es conveniente usar uno de los dos métodos siguientes. El primero consiste en usar `typedef` para acortar los nombres de los tipos, al igual que ya hiciéramos para declarar matrices.

```
typedef map<string, int> MSI;
typedef MSI::iterator MSIit;

int main() {
  MSI m;
  string s;
  while (cin >> s) ++m[s];

  for(MSIit it = m.begin(); it!=m.end(); ++it) {
    cout << it->first << ": "  << it->second << endl;
  }
}
```

La segunda opción consiste en declarar una macro `foreach` que permita escribir bucles como el anterior sin tener que escribir tanto código.

```
#define foreach(i,m) \
for(typeof((m).begin()) i=(m).begin();i!=(m).end();++i)

int main() {
  map<string, int> m;
  string s;
  while (cin >> s) ++m[s];

  foreach(it,m) {
    cout << it->first << ": "  << it->second << endl;
  }
}
```

4.3. Otras operaciones

El criterio de ordenación de las claves de un `map` puede ser modificado del mismo modo como se hacía con las `priority_queue`: usando un tipo propio para el que se ha definido un `operator<`, o añadiendo un tercer parámetro `compara` con el criterio de ordenación en su definición, como en `map<K,T,compara>`. El criterio de comparación sólo puede depender del tipo `K`, y no del tipo `T`. A diferencia de las `priority_queue`, donde el criterio de ordenación era fundamental, en un `map` no suele ser necesario preocuparse del criterio de ordenación, a menos que queramos algún tipo de ordenación concreta para recorrer el `map` con iteradores, o que `K` sea un `struct` definido por nosotros, al que debemos dotar de algún criterio de ordenación porque no tiene ninguno por defecto.

Un `set` es un `map` donde no se guardan datos. Es, por lo tanto, muy parecido. Para insertar una clave `c` en un `set<K> s` hay que usar el método `s.insert(c)` (el `set` no permite el uso del operator de acceso directo `operator[]`). No es un error insertar la misma clave dos veces: la operación no tiene ningún efecto. La otra diferencia radica en el iterador de un `set`: si `it` es un iterador de un `set<K>`, para encontrar el elemento apuntado hay que hacer `*it`, la típica notación para dereferenciar punteros (y no `it->first` o `it->second` como en el caso del `map`).

Por ejemplo, el siguiente código genera un conjunto con todas las potencias de 3 y de 5 menores de 100000.

```
set<int> s;
s.insert(1);
set<int>::iterator it = s.begin();
while(it != s.end()) {
  int k1 = (*it)*3;
  if (k1<100000) s.insert(k1);
  int k2 = (*it)*5;
  if (k2<100000) s.insert(k2);
  ++it;
}
```

En el código anterior se usa que insertar el mismo elemento más de una vez no causa ningún problema. Se usa también otra propiedad, bastante más sutil, de los `set` y los `map`: la inserción o la eliminación de elementos en un `map` o `set` no *invalida* los iteradores que apuntan a los otros elementos. Por ejemplo, la primera vez que se ejecuta el `while`, el iterador `it` apunta al `1`, que es el único elemento del `set`. Si en este momento hiciéramos `++it;`, el iterador `it` pasaría a valer `s.end()`. Sin embargo, cuando acaba esta primera iteración del `while` hemos insertado 2 nuevos elementos en el `set`, por lo que cuando se hace `++it;`, el iterador pasa a apuntar al elemento `3` del conjunto. Observamos pues que, modificando el `set s`, hemos también cambiado el comportamiento del iterador `it`, como se esperaría.

Avanzado. No todos los contenedores de la STL tienen la propiedad de que la inserción o eliminación de elementos no afecta a la validez de los iteradores que apuntan a otros elementos. Por ejemplo, los `vector` no la tienen: hacer `push_back` puede invalidar los iteradores que apuntan a elementos del vector (de hecho, también puede eliminar los punteros o las referencias a los mismos). El motivo es que los iteradores son poco más que punteros, y que algunos contenedores, como el `vector`, pueden requerir *mover* los elementos apuntados de lugar, haciendo que los punteros dejen de ser válidos.

En general, no es aconsejable recorrer un contenedor y modificarlo a la vez, como se ha hecho en el ejemplo anterior, porque no es difícil cometer pequeños errores que vuelvan el código inválido. Por ejemplo, a continuación mostramos dos funciones que intentan eliminar aquellos elementos de un `map` cuya clave tenga menos de 5 letras.

```
typedef map<string, string> MSS;
typedef MSS::iterator MSSit;

//codigo erroneo
void elimina_entradas_pequenas1(MSS &m) {
  for (MSSit it = m.begin(); it != m.end(); ++it) {
    if (it->first.size()<5) m.erase(it);
  }
}

//codigo erroneo (tambien!)
void elimina_entradas_pequenas2(MSS &m) {
  for (MSSit it = m.begin(); it != m.end(); ++it) {
    if (it->first.size()<5) {
      MSSit it2 = it;
      --it;   // (*)
```

```
      m.erase(it2);
    }
  }
}
```

El primer código no es correcto porque, al eliminar el elemento apuntado por el iterador `it`, invalidamos el propio iterador, por lo que perdemos el índice que nos permitía recorrer el `map`. El segundo código intenta arreglar este problema, haciendo que `it` retroceda un elemento antes de borrar la copia `it2`. Sin embargo, el cógido tampoco es correcto: si ocurriera que el primer elemento del `map` tuviera que ser eliminado, la línea marcada por un `(*)` estaría intentando hacer retroceder al iterador `m.begin()`, lo cual provoca error.

Presentamos ahora dos versiones que sí resuelven el problema.

```
//codigo correcto (lento, pero seguro)
void elimina_entradas_pequenas3(MSS &m) {
  vector<MSSit> elimina;
  for (MSSit it = m.begin(); it != m.end(); ++it) {
    if (it->first.size()<5) elimina.push_back(it);
  }
  for (int i=0; i<elimina.size(); ++i) {
    m.erase(elimina[i]);
  }
}

//codigo correcto (tambien)
void elimina_entradas_pequenas4(MSS &m) {
  MSSit it = m.begin();
  while (it != m.end()) {
    if (it->first.size()<5) {
      MSSit it2 = it;
      ++it;
      m.erase(it2);
    }
    else ++it;
  }
}
```

Es obvio que el primer código es, si bien poco elegante, correcto. El segundo código también es correcto: es más eficiente porque no requiere el uso de un vector auxiliar, pero es más complicado de programar. Por ejemplo, notar que hemos tenido que cambiar el `for` por un `while`, para poder controlar el momento exacto en el que incrementamos el iterador `it`: hay que incrementarlo después de hacer la copia `it2`, pero antes de hacer el `erase` del elemento.

Finalmente, acabamos este capítulo explicando un último método del `map` y el `set`, que necesitamos para resolver el problema de encontrar la primera entrada con clave positiva de un `map<int,K>`. Con lo que sabemos podemos recorrer los elementos del `map` uno por uno, empezando por el principio o por el final, hasta encontar el positivo más pequeño; esto, sin embargo, sigue siendo un método lento. Lo que necesitamos es buscar una clave (por ejemplo, el número `0`) y, caso de no existir, obtener un iterador a un elemento *cercano* a la clave buscada, de modo que con unos pocos incrementos o decrementos adicionales podamos encontrar el iterador al elemento buscado. Un método que hace lo que queremos es el `m.lower_bound(c)` que, dada una clave `c`, retorna un iterador a la entrada de clave `c` si ésta existe (al igual que el `find`), pero que, si no existe ninguna entrada con clave `c`, devuelve un iterador a la primera entrada con clave mayor que la clave `c` dada, o `m.end()` si una tal entrada no existe.

Así pues, podemos finalmente resolver el problema `busca_positiva`.

```
//codigo eficiente para encontrar la primera entrada
//con clave positiva de un map, o -1 si no existe ninguna
int busca_positiva(const map<int, string> &m) {
  map<int, string>::iterator it = m.lower_bound(1);
  if (it == m.end()) return -1;
  else return it->first;
}
```

Capítulo 5

Listas: `list<T>`

Una lista es una secuencia de elementos donde, a diferencia de un vector, se puede *insertar* y *eliminar* elementos rápidamente, pero que no puede accederse a ellos según un índice. Uno debe imaginarse una lista como una *cadena* de elementos. Fijado un eslabón de la cadena, y sin importar lo larga que ésta sea, el eslabón podría ser eliminado abriendo y volviendo a cerrar sus eslabones vecinos. De modo similar podríamos insertar un nuevo eslabón, o incluso una nueva cadena, haciendo un número mínimo de operaciones. Por contra, para desplazarnos por la cadena nos tendríamos que mover eslabón a eslabón hasta llegar al que quisiéramos. Esta analogía permite entender que operaciones son lentas en una lista (esencialmente, desplazarse por la misma) y cuales son rápidas (insertar o eliminar elementos o incluso listas enteras, siempre que se conozcan los puntos donde deben realizarse los "cortes").

Avanzado. La analogía anterior es correcta porque el tipo `list` de la STL se implementa, como uno esperaría, con nodos formando una cadena de punteros doblemente enlazada. Por lo tanto, todas las operaciones tienen tiempo constante excepto, por supuesto, desplazarse por la lista, que tarda tiempo proporcional al número de elementos que se avance.

5.1. Operaciones básicas

Si `l` es una `list<T>`, se tiene que `l.push_back(e)` y `l.push_front(e)` insertan un elemento `e` de tipo `T` al final o al principio de la lista. Para obtener el primer elemento de la lista se llama a `l.front()`, y para obtener el último a `l.back()`; para eliminar el primer o el último elemento se llama a `l.pop_front()` o `l.pop_back()`. En este sentido, una lista puede actuar como una `queue` o como una `pila` al mismo tiempo.

¡Cuidado! Como de habitual, hay que tener cuidado en no preguntar por el `back()` o el `front()` de una lista vacía.

Al igual que muchos otros contenedores de la STL, podemos escribir `l.empty()` para saber si la lista está vacía, `l.size()` para conocer el número de elementos, y `l.clear()` para borrarla.

Avanzado. A diferencia de los otros contenedores, la `list` tiene la desagradable propiedad de que `l.size()` no tarda tiempo constante en responder, sino tiempo lineal: la implementación recorre la lista entera para contar cuantos elementos hay. Por lo tanto, es conveniente usar el método `size()` lo menos posible. En su lugar, podemos preguntar si la lista está vacía con el `l.empty()` (esta operación sí tarda tiempo constante), o guardar el `size()` de la lista en una variable, y contabilizar nosotros mismos el número de elementos que tiene la lista.

5.2. Operaciones con iteradores

Para trabajar con los elementos *interiores* de la lista es necesario usar *iteradores*, que funcionan de modo parecido a los iteradores de un `map` o de un `set`. El método `l.begin()` devuelve un iterador al primer elemento de la lista, y `l.end()` devuelve un iterador que apunta una posición después del último. Para hacer que un iterador `it` se desplace por la lista usamos los operadores `++it` y `--it`; podemos recuperar el elemento apuntado por el iterador escribiendo `*it`. Para *eliminar* un elemento de la lista necesitamos un iterador `it` que apunte al mismo, y llamar a `l.erase(it)`. Para *insertar* un elemento `e` necesitamos un iterador `it` que apunte al que será el elemento *siguiente* al que queremos insertar, y escribir `l.insert(it,e)`: de este modo, podemos insertar al principio si `it==l.begin()` y al final cuando `it==l.end()`.

¡Cuidado! Al igual que con un `map` o un `set`, está prohibido llamar a `--it` si `it` es igual a `l.begin()`, y llamar a `++it`, `*it` o `l.erase(it)` si `it` es igual a `l.end()`. Cuando se borra un elemento de una lista, cualquier iterador que apuntase al mismo se invalida y no debe volver a usarse. Los iteradores restantes continúan siendo válidos.

En general, escribir programas que modifiquen listas es delicado, porque se hace necesario trabajar con varios iteradores, tener en cuenta como afecta a los recorridos el hecho de añadir o eliminar elementos, y vigilar de no hacer operaciones inválidas. Mostramos, como ejemplo, una función que, a partir de una secuencia de elementos, elimina aquellos que no son más grandes que todos los anteriores. Por ejemplo, para la secuencia $1, 3, 4, 2, 4, 7, 7, 1$, nos quedaría la secuencia $1, 3, 4, 7$.

```
typedef list<int>::iterator LIit;
void subsecuencia(list<int> &l) {
  LIit it = l.begin();
  while (it != l.end()) {
    LIit next = it;
    ++next;
    if (next != l.end() and *it>=*next) {
      l.erase(next);
    }
    else {
      it = next;
    }
  }
}
```

Cuando escribimos código que trabaja con listas vale la pena intentar comprobar que no se generan llamadas inválidas. Por ejemplo, podemos hacer `++next` porque sabemos que en ese punto `next==it` y que `it!=l.end()`, y podemos escribir `*next` y `l.erase(next)` porque sabemos que en ese punto `next!=l.end()`.

5.3. Otras operaciones

La STL ofrece algunos algoritmos programados específicamente para listas que pueden ser útiles. Podemos *girar* una lista llamando a `l.reverse()`, o eliminar elementos repetidos que aparezcan consecutivamente llamando a `l.unique()`. Si tenemos dos listas `l1` y `l2` ordenadas, podemos *fusionar* la segunda en la primera, obteniendo una lista que contenga todos los elementos ordenados, llamando a `l1.merge(l2)`. (La lista `l2` se borra al hacer la llamada). Por último, podemos *ordenar* una lista `l`

de tipo `T` llamando `l.sort()`, o `l.sort(compara)` si queremos usar alguna función de comparación `bool compara(const T &a, const T &b)` (siguiendo el formato descrito para ordenar un `vector`).

¡Cuidado! Escribir `sort(l.begin(), l.end())`, como se hacía para ordenar un `vector`, no funcionará, dando lugar a un error de compilación bastante largo. Es necesario escribir `l.sort()`.

Avanzado. El motivo de que no funcione es que la implementación del `sort` necesita poder hacer *aritmética* con los iteradores, como por ejemplo `it + c` donde `c` es un `int` para apuntar al elemento `c` posiciones más allá del iterador `it`, o `it2-it1` para obtener el número de elementos que hay entre el iterador `it2` y el `it1`. Los iteradores de una lista no permiten realizar este tipo de operaciones: el error de compilación se da en el código interno del `sort`, cuando se ve que no se puede compilar el código templatizado con iteradores de lista. Por este motivo se añadió un método `sort()` que implementara el algorismo de ordenación *mergesort*.

Las listas ofrecen la posibilidad única de romper una lista en varias sublistas o de juntar varias listas en una realizando únicamente un número pequeño y fijo de operaciones, sin importar lo largas que fueran las listas. Esto es fácil de visualizar si se entiende que una lista funciona como una *cadena* donde los elementos hacen de *eslabones*, y que puede abrirse y cerrarse en cada eslabón. La única condición necesaria para poder realizar estas operaciones es que no podemos *crear* ni *eliminar* elementos. Por ejemplo, podemos *juntar* dos listas en una sola, pero al hacerlo obtenemos una lista con todos los elementos: las listas originales se pierden. Si quisiéramos mantener una copia alguna de ellas, entonces deberíamos haberla hecho antes de juntarlas, y el tiempo que se tardara en hacer la copia dependería del número de elementos que hay que copiar. De modo similar, podemos *extraer* una sublista de dentro de otra, pero al hacerlo se mantiene en memoria la lista extraída, desconectada de la original. Si no hacemos nada con ella se recuperará la memoria que ésta ocupa, y la propia lista, al borrarse, deberá recorrer los elementos uno por uno.

La operación que permite hacer este tipo de operaciones con listas es el método `splice`. Si `l1` y `l2` son listas, `it1` es un iterador de la lista `l1`, y `it2b` y `it2e` son iteradores de la lista `l2`, la llamada `l1.splice(it, l2, it2b, it2e)` *recorta* de la lista `l2` el fragmento comprendido entre `it2b` (inclusive) y `it2e` (no inclusive), y lo inserta justo antes del elemento apuntado por `it1` en la lista `l1`. Veamos unos ejemplos.

```
// juntar: l1 = l1,l2
l1.splice(l1.end(), l2, l2.begin(), l2.end());

// juntar: l1 = l2,l1
l1.splice(l1.begin(), l2, l2.begin(), l2.end());

// romper:
//   de l1=la,lb, l2=vacia, itb apunta a lb en l1;
//   obtener l1=la, l2=lb
l2.splice(l2.begin(), l1, itb, l1.end());
```