La Standard Template Library (STL) es una colección de estructuras de datos genéricas y algoritmos escritos en C++. STL no es la primera de tales librerías, así la mayor parte de los compiladores de C++ disponen (o disponían) de librerías similares y, también, están disponibles varias librerías comerciales. Uno de los problemas de estas librerías es que son mutuamente incompatibles, lo que obliga a los programadores a aprender nuevas librerías y a migrar de un proyecto a otro y de uno a otro compilador. Sin embargo, STL ha sido adoptado por el comité ANSI de estandarización del C++, lo que significa que está (o estará) soportado como una extensión más del lenguaje por todos los compiladores.

El diseño de la Standard Template Library es el resultado de varios años de investigación dirigidos por Alexander Stepanov y Meng Lee de Hewlett-Packard, y David Musser del Rensselaer Polytechnic Institute. Su desarrollo se inspiró en otras librerías orientadas a objetos y en la experiencia de sus creadores en lenguajes de programación imperativos y funcionales, tales como Ada y Scheme.

La STL proporciona una colección de estructuras de datos contenedoras y algoritmos genéricos que se pueden utilizar con éstas. Una estructura de datos se dice que es **contenedora si puede contener instancias de otras estructuras de datos.** En concreto, **la STL dispone de las estructuras indicadas en la tabla** siguiente:

|  |  |  |
| --- | --- | --- |
| **Contenedores lineales** | **Contenedores asociativos** | **Contenedores adaptados** |
| **Vector**  *vector<T>* | **Conjunto**  *set<T, Compare>* | **Pila**  *stack<Container>* |
| **Lista**  *vector<T>* | **Multiconjunto**  *multiset<T, Compare>* | **Cola**  *queue<Container>* |
| **Doble cola**  *deque<T>* | **Aplicación**  *map<Key, T, Compare>* | **Cola de prioridad**  *priority\_queue<Container,Compare>* |
|  | **Multiaplicación**  *multimap<Key, T, Compare>* |  |

Estos contenedores se dice que son genéricos porque pueden contener instancias de cualquier otro tipo de dato, para lo que utilizan de forma extensiva los *templates* de C++.

Los contenedores se dividen en tres categorías:

      **Contenedores lineales**. Almacenan los objetos de forma secuencial permitiendo el acceso a los mismos de forma secuencial y/o aleatoria en función de la naturaleza del contenedor.

      **Contenedores asociativos**. En este caso cada objeto almacenado en el contenedor tiene asociada una clave. Mediante la clave los objetos se pueden almacenar en el contenedor, o recuperar del mismo, de forma rápida.

      **Contenedores adaptados**. Permiten cambiar un contenedor en un nuevo contenedor modificando la *interface* (métodos públicos y datos miembro) del primero. En la mayor parte de los casos, el nuevo contenedor únicamente requerirá un subconjunto de las capacidades que proporciona el contenedor original.

La adopción de STL ofrece varias ventajas:

     Al ser estándar, está (o estará) disponible por todos los compiladores y plataformas. Esto permitirá utilizar la misma librería en todos los proyectos y disminuirá el tiempo de aprendizaje necesario para que los programadores cambien de proyecto.

     El uso de una librería de componentes reutilizables incrementa la productividad ya que los programadores no tienen que escribir sus propios algoritmos. Además, utilizar una librería libre de errores no sólo reduce el tiempo de desarrollo, sino que también incrementa la robustez de la aplicación en la que se utiliza.

     Las aplicaciones pueden escribirse rápidamente ya que se construyen a partir de algoritmos eficientes y los programadores pueden seleccionar el algoritmo más rápido para una situación dada.

     Se incrementa la legibilidad del código, lo que hará que éste sea mucho más fácil de mantener.

     Proporciona su propia gestión de memoria. El almacenamiento de memoria es automático y portátil, así el programador ignorará problemas tales como las limitaciones del modelo de memoria del PC.

***Componentes de STL***

La librería STL dispone de cinco componentes diferentes:

|  |  |
| --- | --- |
| **Algoritmos** | la STL proporciona una amplia variedad de algoritmos comunes entre los que se incluyen los de ordenación, búsqueda y algoritmos numéricos. Los algoritmos se pueden utilizar con estructuras de datos de la librería, vectores, o estructuras de datos definidas por el usuario y provistas de iteradores |
| **Iteradores** | una generalización del puntero que permite al programador visitar cada elemento de una estructura de datos contenedora. |
| **contenedores** | estructuras de datos que contienen, y permiten manipular, otras estructuras de datos. |
| **funciones objeto** | varios de los algoritmos proporcionados por la STL, permiten pasar una función al algoritmo para adecuar su funcionalidad. Las funciones objeto son una generalización del puntero a la función del C |
| **adaptadores** | toman un componente y le proporciona una *interface* diferente. Esto permite transformar una estructura de dato en otra que tiene las mismas facilidades pero cuya *interface* proporciona un conjunto de operaciones diferente. |

***Algoritmos genéricos***

Habitualmente los programadores escriben código que frecuentemente reescriben para utilizarlo en otros programas, ya que en éstos se emplean una y otra vez numerosos algoritmos comunes (algoritmos de búsqueda, algoritmos de ordenación,...) pero que deben adaptarse a cada problema. Es obvio que está es una tarea repetitiva que debe ser eliminada, para lo que debemos producir algoritmos genéricos que puedan ser utilizados en una amplia variedad de situaciones.

Así, existe una gran cantidad de librerías de código reutilizable. Los ejemplos más comunes y extendidos son las librerías de funciones matemáticas y las de funciones de manipulación de cadenas de caracteres. De hecho, los compiladores proporcionan de forma habitual este tipo de librerías. La cuestión es, por qué éste tipo de librerías tiene éxito mientras que otro tipo de librerías no.

La razón es que tanto las funciones matemáticas como las de manipulación de cadenas de caracteres trabajan con un número  muy limitado de tipos de datos. La mayor parte de las funciones matemáticas con enteros y valores de punto flotante y las funciones de manipulación de cadenas de caracteres con una representación de éstas. De manera, que al estar limitado el rango de tipos a utilizar el código fuente de las funciones es fácilmente abordable. Los algoritmos no tienen que rescribirse para trabajar con otros tipos de datos, ya que no es aplicable ningún otro tipo de dato. Por tanto, podemos concluir que para producir algoritmos genéricos debemos ser capaces de separar los algoritmos de la representación de datos que manipulan.

***Iteradores***

Una de las operaciones más comunes sobre una estructura de datos contenedora es la de recorrer todos o algunos de los elementos almacenados en ella según un orden predefinido. Un iterador es un objeto que puede retornar por turno una referencia a cada uno de los elementos de un contenedor.

Los iteradores son la base de construcción de los algoritmos genéricos. Así, por ejemplo, podríamos escribir el siguiente algoritmo genérico de búsqueda lineal

template <class T>

T \* linearSearch (T \* first, T \* last, T & value)

{

for (; first != last; first++)

       if ((\*first) == value) **return** first;

**return** 0;

}

Puesto que un iterador permite retornar una referencia a cada uno de los elementos de una estructura de datos contenedora, podemos como un puntero. Es más, un iterador debería soportar todas las operaciones que puedan realizarse con un puntero. Estas operaciones se resumen en la tabla siguiente para los iteradores *x* e *y* y el entero *n*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *x++* | *x + n* | *x - y* | *x > y* | *\*x* |
| *++x* | *x – n* | *x == y* | *x <= y* | *x = y* |
| *x--* | *x += n* | *x != y* | *x >= y* |  |
| *--x* | *x -= n* | *x < y* | *x[n]* |  |

Si somos capaces de diseñar un iterador que soporte las mismas operaciones que un puntero, se podrá utilizar como tal, y ambos serán indistinguibles. Esto es exactamente lo que los diseñadores del STL han hecho, han creado un iterador que generaliza el concepto del puntero y con el que se pueden realizar las mismas operaciones. El resultado es que en cualquiera de los algoritmos de STL que utilicen iteradores también se puede utilizar punteros.

Como uno de los objetivos de la STL es ser eficiente, no todos los iteradores soportan todas las operaciones. Los iteradores se dividen en varias categorías de forma que en cada una de ellas únicamente se proporcionan las operaciones que pueden implementarse de forma eficiente. Cada una de las clases contenedoras soporta una o más tipos de iteradores dependiendo de la eficiencia de las operaciones del iterador para cada contenedor. Las categorías de iteradores de la STL son las siguientes:

|  |  |
| --- | --- |
| **forward iterators** | Iteradores que pueden avanzar al elemento siguiente. |
| **bidirectional iterators** | Iteradores que pueden avanzar al elemento siguiente o retroceder al elemento. |
| **random access iterators** | Iteradores que pueden avanzar o retroceder más de una posición de una vez. |

Además de éstos, la STL proporciona otras dos clases de iteradores que permiten leer datos desde un *stream* a un contenedor y escribir datos desde un contenedor hacia un *stream*. Los diseñadores de la STL pensaron que si se asociaban iteradores a los *streams*, la lectura y escritura de datos podría ser más fácilmente manejable con las funciones proporcionadas por la STL. Por otra parte, los iteradores asociados a los *streams* no pueden tener todas las operaciones soportadas por los otros tipos de iteradores. Por ejemplo, una vez que se ha escrito un dato en un *stream* no es posible retroceder y escribir otro sobre él. Las dos clases iteradoras adicionales son las siguientes:

|  |  |
| --- | --- |
| **input iterators** | Iteradores que pueden mover una posición hacia adelante en un tiempo y permiten dereferenciar cuando se utilizan como *r-value* (en la parte derecha de una asignación). |
| **output iterators** | Iteradores que pueden mover una posición hacia delante en un tiempo y permite dereferenciar cuando se utilizan como *l-value*(en la parte izquierda de una asignación). |

Virtualmente todos los algoritmos de la STL utilizan iteradores para indicar el rango de operadores sobre el que debe operar. De forma similar, todos los contenedores de la STL proporcionan iteradores para permitir el acceso a los objetos que contiene. Así, comprender los iteradores es vital para utilizar de forma efectiva la STL.

Input iterators

Los iteradores de entrada sólo se pueden mover hacia delante y sólo pueden utilizarse para recuperar valores, no para valores de salida. Los iteradores de entrada *a* y *b* soportan las siguientes operaciones:

|  |  |  |
| --- | --- | --- |
| *a++* | *++a* | *\*a*  (sólo como *r-value*) |
| *a == b* | *a != b* |  |

Los iteradores de entrada tienen la propiedad de que si *a* y *b* son iteradores de entrada sobre un mismo objeto contenedor, y *a == b*, esto no implica que *++a == ++b*. Esto es porque el principal propósito de los operadores de entrada es utilizarlos como clase base de *input streams iterators*. Los *stream iterators* permiten manipular *streams* por medio de iteradores y, por tanto, deben tenerse en cuenta las restricciones impuestas por los propios *streams*.

Output iterators

Los iteradores de salida, como los de entrada, únicamente se pueden mover hacia delante pero difieren en que sólo se pueden dereferenciar para asignar un valor, no para recuperarlo. Un operador de salida *a* soporta las operaciones:

|  |  |  |
| --- | --- | --- |
| *++a* | *a++* | *\*a*(sólo como *l-value*) |

Como los iteradores de entrada, los de salida también se utilizan como clase base para *output streams iterators*.

Forward iterators

Los iteradores hacia delante están diseñados para recorrer contenedores cuyos valores puedan ser añadidos y recuperados. Esto relaja algunas de las restricciones de los iteradores de entrada/salida pero mantienen la restricción de que únicamente se pueden mover hacia delante. Los iteradores hacia delante *a* y *b* soportan las operaciones:

|  |  |  |
| --- | --- | --- |
| *A++* | *++a* | *\*a* |
| *A = b* | *a == b* | *a != b* |

Los iteradores hacia delante pueden dereferenciarse para asignar un valor al objeto referenciado, o bien, para recuperar su valor. Un iterador puede asignar el valor de otro, lo que implica que se referirán al mismo objeto.

Bidirectional iterators

Los iteradores bidireccionales eliminan la restricción de los anteriores en los que sólo se puede avanzar hacia delante. Los iteradores bidireccioales *a* y *b* soportan las operaciones:

|  |  |  |  |
| --- | --- | --- | --- |
| *a++* | *++a* | *a--* | *--a* |
| *\*a* | *a = b* | *a == b* | *a != b* |

Random access iterators

Los iteradores de acceso aleatorio eliminan la restricción de que un iterador sólo puede avanzar al siguiente elemento, o retroceder al elemento previo, del contenedor en una única operación. Los iteradores de acceso aleatorio *a* y *b*, más el entero *n*, soportan las operaciones:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *a++* | *a + n* | *a – b* | *a > b* | *\*a* |
| *++a* | *a – n* | *a == b* | *a <= b* | *a = b* |
| *a--* | *a += n* | *a != b* | *a >= b* |  |
| *--a* | *a -= n* | *a < b* | *a[n]* |  |

Un iterador de acceso aleatorio puede incrementarse o decrementarse por un valor entero *n*y esto es equivalente a incrementar o decrementar el iterador por uno *n* veces. El operador de índice *a[n]* permite retornar el objeto referenciado por el iterador *n* posiciones hacia delante del iterador *a*.

***Ejemplos de uso de iteradores***

Los iteradores se pueden utilizar de la misma forma que se utilizan los punteros sobre los vectores pero, además, incorporan una mayor funcionalidad de forma que se pueden utilizar sobre cualquier estructura de datos contenedora definida por la STL. Se verán aquí algunos ejemplos sencillos de cómo se utilizan los iteradores en los algoritmos de la STL y contenedores.

Sream iterators

Se utilizarán aquí los tipos de iteradores *istream\_iterator* y *ostream\_iterator*, que como ya se indicó derivan de los iteradores de entrada y de salida, respectivamente. De forma, que cuando se utilizan se pueden asociar con cualquier *stream* de entrada abierto o *stream* de salida.

En el primer ejemplo se muestra como se puede copiar el contenido de un *stream* de entrada a un *stream* de salida mediante iteradores (sin utilizar los *streams* directamente).

#include <iterator.h>

**template** <**class** InputIterator, **class** OutputIterator>

void copyToEnd (InputIterator start, InputIterator end, OutputIterator out)

{

  while (start != end) {

    \*out++ = \*start++;

  }

}

main ()

{

  istream\_iterator<int, ptrdiff\_t> int\_in(cin);

  ostream\_iterator<int> int\_out(cout," ");

  copyToEnd(int\_in, istream\_iterator<int, ptrdiff\_t>(), int\_out);

  cout << "\nEOF\n";

}

A la función copyToEnd se le pasa un iterador indicando el comienzo del *stream* de entrada, un iterador que indica la posición siguiente a la última del *stream* de salida y un iterador que referencia el *stream* al que se copia.

La función main() contiene las declaraciones de los iteradores de entrada/salida y la llamada a copyToEnd. Tanto la declaración del iterador de entrada como la del de salida llevan como parámetro el tipo de valor (int) de entrada o salida. El *template* para istream\_iterator requiere un parámetro extra, el tipo de distancia entre objetos en el *stream*, el cual siempre es ptrdiff\_t.

El constructor para los iteradores de entrada acepta un único parámetro: una referencia al *stream* que debe manipular. En cuanto al constructor para el iterador de salida además de la referencia al *stream* de salida puede indicársele, opcionalmente, la cadena de caracteres que se imprimirá entre cada par de valores. En nuestro caso, el separador entre cada par de valores enteros es un espacio en blanco.

A la llamada de la función copyToEnd debe pasársele un iterador de entrada que indique la posición siguiente a la última del *stream* de entrada, éste valor se obtiene mediante la llamada al constructor por defecto de la clase istream\_iterator.

En los ejemplos siguientes, veremos como se pueden utilizar los iteradores de entrada/salida en conjunción con funciones de la STL que requieren como parámetros iteradores.

En el primero de ellos se utiliza la función accumulate para una serie de valores enteros que se introducen por la entrada estándar. La función acepta dos iteradores que delimiten el rango de elementos a sumar y un valor inicial, y retorna el resultado de sumar el rango de elemento y el valor inicial.

El siguiente ejemplo utiliza la función copy para copiar a la salida estándar las componentes de un vector de cadenas de caracteres. La función copy copia todos los valores delimitados por los dos primeros iteradores al objeto referenciado por el tercer iterador. En nuestro caso, los dos primeros son las posiciones (punteros) inicial y siguiente a la última del vector y el tercer argumento es un iterador de salida que referencia la salida estándar.

#include <algo.h>

main ()

{

  /\* *suma los enteros dados por la entrada estándar* \*/

  istream\_iterator<int, ptrdiff\_t> intIn(cin);

  int sum;

  sum = accumulate(intIn, istream\_iterator<int, ptrdiff\_t>(), 0);

  cout << "suma = " << sum << endl;

}

#include <algo.h>

main ()

{

  ostream\_iterator<char\*> strOut(cout, "\n");

**const** int len = 3;

  char\* text[len] = {"hello", "and", "goodbye"};

  copy(text, text+len, strOut);

}

Forward iterators

Los iteradores hacia delante se utilizan para recorren una secuencia avanzando un elemento de cada vez.

En el primer ejemplo se define y utiliza la función add10 para sumar 10 a todos los elementos de un contenedor, concretamente, de un vector de enteros.

#include <algo.h>

**template** <**class** ForwardIterator>

void add10 (ForwardIterator first, ForwardIterator last)

{

  while (first != last) {

    \*first = \*first + 10;

    first++;

  }

}

main ()

{

**const** int len = 15;

  int data[len];

  ostream\_iterator<int> outStream(cout, " ");

  /\* *inicialización del vector* \*/

  for (unsigned int i=0; i<len; i++)

    data[i] = i;

  /\**suma 10 a cada componente y escribe el vector*\*/

  add10(data, data+len);

  copy(data, data+len, outStream);

  cout << endl;

}

Buscar el comienzo y la siguiente posición a la última del vector es relativamente fácil. Para los contenedores de STL esto no es tan evidente y para ello se proporcionan los métodos begin() y end(), respectivamente. El ejemplo siguiente, muestra como utilizar estos métodos sobre un iterador que referencia una lista. La lista es un contenedor de la STL que está implementada como una lista doblemente enlazada.

#include <list.h>

main ()

{

  list<int> l;

  list<int>::iterator lItr;

  ostream\_iterator<int> outStream(cout, " ");

  /\* *inicialización de la lista l: 0 1 2 3 4 5 6 7 8 9* \*/

  for (unsigned int i=0; i<10; i++)

    l.push\_back(i);

  /\* *escribe la lista l* \*/

  copy(l.begin(), l.end(), outStream);

  cout << endl;

  /\* *inserta 100 en la segunda posicion de la lista l* \*/

  lItr = l.begin();

  lItr++;

  l.insert(lItr, 100);

  /\* *otra forma de escribir la lista l* \*/

  lItr = l.begin();

  while (lItr != l.end())

    cout << \*lItr++ << ' ';

  cout << endl;

}

El programa comienza declarando una lista de enteros y un iterador para una lista de enteros. También se declara un ostream\_iterator que referencia la salida estándar. El bucle for inicializa la lista que contendrá los valores enteros de 0 a 9, añadiéndolos en este orden al final de la lista.

Algunos algoritmos STL y estructuras de datos utilizan los iteradores para indicar un miembro específico del contenedor. Por ejemplo, esto permite insertar en una lista antes de un determinado miembro de la lista, tal y como ocurre en el programa de ejemplo cuando se inserta el valor 100 en la segunda posición de la lista.

Bidirectional iterators

Un iterador bidireccional tiene todas las propiedades de los iteradores hacia delante más la facilidad de retroceder una posición decrementándolo. Este tipo de iteradores es muy utilizado en algoritmos que requieren recorrer varias veces los datos o en los que éstos deben procesarse en orden inverso.

Hay dos tipos de iteradores bidireccionales bidirectional\_iterator y reverse\_bidirectional\_iterator, los cuales se obtienen aplicando un adaptador a un iterador bidireccional. Ambos tipos tienen la capacidad para moverse en dos direcciones (hacia delante y hacia atrás), un bidirectional\_iterator se utiliza para procesar los datos hacia delante desde begin() hasta end(). Un reverse\_bidirectional\_iterator se utiliza para procesar los datos hacia atrás desde rbegin() hasta rend(). La diferencia está en como sé inicializan los iteradores y en como se determina cuando se alcanza el final de la secuencia de elementos almacenados en el contenedor.

Si se quieren procesar los datos hacia delante el iterador debe referenciar inicialmente el primer elemento del contenedor y terminar cuando se alcanza la posición siguiente al último elemento del contenedor, estos valores son los que retornan los métodos begin() y end(). Procesar los datos en sentido inverso requiere que el iterador referencie inicialmente el último elemento del contenedor y terminar cuando se alcanza la posición anterior a la del primer elemento del contenedor, estos valores son los que retornan los métodos rbegin() y rend(). Para incrementar la seguridad de los iteradores, se proporcionan los dos tipos de iteradores y los métodos de inicialización retornan el correspondiente tipo de iterador. Así, begin() y end() retornan bidirectional\_iterators y rbegin() y rend()  retornan reverse\_bidirectional\_iterators.

En el ejemplo siguiente se muestra como trabajan ambos tipos de iteradores.

#include <list.h>

main ()

{

  list<int> l;

  list<int>::iterator lItr;

  list<int>::reverse\_iterator lRitr;

  ostream\_iterator<int> outStream(cout, " ");

  /\* *inicialización de la lista l: 1 2 3 4 5 6 7 8 9* \*/

  for (unsigned int i=1; i<10; i++)

    l.push\_back(i);

  /\**escribe la lista l* \*/

  copy(l.begin(), l.end(), outStream);

  cout << endl;

  /\* *escribe la lista l en orden inverso* \*/

  lItr = l.end();

  while (lItr != l.begin())

    cout << \*lItr-- << ' ';

  cout << endl;

  /\* *resultado erróneo: ? 9 8 7 6 5 4 3 2* \*/

  lRitr = l.rbegin();

  while (lRitr != l.rend())

    cout << \*lRitr++ << ' ';

  cout << endl;

  /\* *resultado correcto: 9 8 7 6 5 4 3 2 1* \*/

}

Random access iterators

Los iteradores de acceso aleatorio tienen todas las capacidades de los iteradores bidireccionales más la facilidad para realizar movimientos de más de una posición y la habilidad para comparar las posiciones de dos iteradores que referencian un mismo objeto.

Para mostrar el funcionamiento de este tipo de iteradores introduciremos un nuevo contenedor que soporta accesos aleatorios, el vector. El vector es similar al tipo predefinido *array* pero en el que la dimensión se puede modificar de forma dinámica. Un vector se construye e inicializa del mismo modo que una lista, como se muestra en el ejemplo siguiente

#include <vector.h>

main ()

{

  vector<int> v;

  vector<int>::iterator vItr;

  vector<int>::reverse\_iterator vRitr;

  ostream\_iterator<int> outStream(cout, " ");

  /\* *inicialización del vector v: 0 1 2 3 4 5 6 7 8 9* \*/

  for (unsigned int i=0; i<10; i++)

    v.push\_back(i);

  /\* *inserta 99 en la cuarta posición y escribe el vector* \*/

  vItr = v.begin();

  vItr += 3; /\* *avanza tres posiciones* \*/

  v.insert(vItr, 99);

  copy(v.begin(), v.end(), outStream);

  cout << endl;

  /\* *inserta 88 en la cuarta posición del*

*vector v sin modificar el iterador* \*/

  vItr = v.begin();

  v.insert(vItr+3, 88);

  copy(v.begin(), v.end(), outStream);

  cout << endl;

  /\* *acceso a los elementos mediante el operador de índice* \*/

  vItr = v.begin();

  for (unsigned int i=0; i<5; i++)

    cout << vItr[i] << ' ';

  cout << endl;

  /\* *escritura del vector v en orden inverso* \*/

  vRitr = v.rbegin();

  while (vRitr != v.rend())

    cout << \*vRitr++ << ' ';

  cout << endl;

  /\* *muestra de comparación de iteradores* \*/

  vector<int>::iterator itr;

  itr = v.begin() + 5;

  vItr = v.begin();

  while (vItr <= itr)

    cout << \*vItr++ << ' ';

  cout << endl;

}

Iterator functions

Únicamente los iteradores de acceso aleatorio están provistos de las operaciones operator+ y operator-, ya que sólo en este tipo de operadores estas operaciones son de tiempo constante. La función advance permite mover los iteradores de entrada, hacia delante y bidireccionales más de una posición. Esta función se realiza aplicando varias veces los operadores operator++ u operator--, por lo que tiene un coste lineal; es decir, que es una función iteradora.

La función advance toma dos parámetros: el iterador a mover y la distancia que debe moverse. La distancia puede ser negativa sólo si el iterador es bidireccional o de acceso aleaotorio.

**template** <**class** InputIterator, **class** Distance>

inline void advance (InputIterator & itr, Distance n);

A veces es necesario buscar la distancia entre dos iteradores que referencian un mismo objeto. La función distance permite realizar dicha búsqueda.

**template** <**class** InputIterator, **class** Distance>

inline void distance (InputIterator first, InputIterator last, Distance & n);

**Tipos de datos**

***Vectores***

Un vector está implementando con como un bloque de memoria contiguo de forma similar a un *array*. Las características de un vector incluyen:

       Inserciones y borrados en tiempo constante al principio y al final.

       Inserciones y borrados de coste lineal en posiciones intermedias.

       Gestión automática de memoria.

Cuando se crea el vector se asigna espacio para los elementos y si es necesario se redimensiona, automáticamente, el tamaño del vector.

**Operaciones para el tipo de dato vector**

|  |  |  |
| --- | --- | --- |
| *Constructores* |  |  |
| vector<T> v; | Constructor por defecto | *O(1)* |
| vector<T> (int, T) | Constructor con tamaño y valor inicial dados | *O(n)* |
| vector<T> v (aVector); | Constructor de copia | *O(n)* |
| *Acceso a elementos* |  |  |
| v[i] | Acceso por índice, también puede asignarse | *O(1)* |
| v.front() | Primer valor de la colección | *O(1)* |
| v.back() | Último valor de la colección | *O(1)* |
| *Inserción* |  |  |
| v.push\_front (T) | Añade un elemento al principio del vector | *O(1)* |
| v.push\_back (T) | Añade un elemento al final del vector | *O(1)* |
| v.insert (iterator, T) | Inserta un nuevo elementos antes del iterador | *O(n)* |
| v.swap (vector<T>) | Intercambia valores con otro vector | *O(n)* |
| *Borrado* |  |  |
| v.pop\_front () | Borra el primer elemento del vector | *O(1)* |
| v.pop\_back () | Borra el último elemento del vector | *O(1)* |
| v.erase (iterator) | Borra el elemento indicado por el iterador | *O(n)* |
| v.erase (iterator, iterator) | Borra un rango de valores | *O(n)* |
| *Tamaño* |  |  |
| v.capacity () | Número máximo de elementos del *buffer* | *O(1)* |
| v.size () | Número de elementos en el vector | *O(1)* |
| v.resize (unsigned, T) | Cambia el tamaño, rellenando con un valor | *O(n)* |
| v.reserve (unsigned) | Pone el tamaño del *buffer* | *O(n)* |
| v.empty () | Cierto si el vector está vacío | *O(1)* |
| *Iteradores* |  |  |
| vector<T>::iterator itr | Declara un nuevo iterador | *O(1)* |
| v.begin () | Iterador que referencia al primer elemento | *O(1)* |
| v.end () | Iterador que referencia al siguiente al último | *O(1)* |
| vector<T>::reverse\_iterator ritr | Declara un nuevo reverse\_iterator |  |
| v.rbegin () | Reverse\_iterator que referencia al último elemento | *O(1)* |
| v.rend () | Reverse\_iterator que referencia al anterior al primero | *O(1)* |

**Algoritmos genéricos útiles con vectores**

|  |
| --- |
| fill (itetator start, iterator stop, value)  Rellena el vector con un valor dado |
|  |
| copy (iterator start, iterator stop, iterator destination)  Copia una secuencia en otro |
|  |
| max\_element (iterator start, iterator stop)  Busca el mayor valor de la colección |
|  |
| min\_element (iterator start, iterator stop)  Busca el menor valor de la colección |
|  |
| reverse (iterator start, iterator stop)  Invierte los elementos en la colección |
|  |
| count (iterator start, iterator stop, value, counter)  Cuenta el número de elementos que coinciden con el valor e incrementa el contador |
|  |
| count (iterator start, iterator stop, unary function counter)  Cuenta el número de elementos que satisfacen la función e incrementa el contador |
|  |
| transform (iterator start, iterator stop, iterator destination, unary function)  Transforma los elementos utilizando la función y los situa en el destino indicado |
|  |
| tind (iterator start, iterator stop, value)  Busca un valor en la colección retornado un iterador a éste, si no lo encuentra retorna stop |
|  |
| find\_if (iterator start, iterator stop, unary function)  Busca un valor que satisfaga la función retornado un iterador, si no lo encuentra retorna stop |
|  |
| replace (iterator start, iterator stop, value, replacement value)  Reemplaza todas las ocurrencias del valor con el valor de reemplazamiento |
|  |
| replace (iterator start, iterator stop, unary function, replacement value)  Reemplaza todas las ocurrencias que satisfacen la función con el valor de reemplazamiento |
|  |
| sort (iterator start, iterator stop)  Coloca los elementos en orden ascendente |
|  |
| for\_each (iterator start, iterator stop, function)  Aplica la función a cada elemento de la colección |
|  |
| iter\_swap (iterator, iterator)  Intercambia los valores especificados por ambos iteradores |

**Algoritmos genéricos útiles con vectores ordenados**

|  |
| --- |
| merge (itetator start1, iterator stop1, iterator start2, iterator stop2, iterator dest)  Mezcla dos colecciones ordenadas en una tercera |
|  |
| inplace\_merge (iterator start, iterator center, iterator destination)  Mezcla dos secuencias ordenadas y adyacentes en una |
|  |
| binary\_search (iterator start, iterator stop, value)  Busca el valor en la colección, retorna un booleano |
|  |
| lower\_bound (iterator start, iterator stop, value)  Busca la primera ocurrencia de un elemento mayor o igual al valor, retorna un iterador |
|  |
| upper\_bound (iterator start, iterator stop)  Busca la primera ocurrencia de un elemento estrictamente mayor que valor, retorna un iterador |

***Listas***

El tipo de dato list<T> se presenta como una lista doblemente enlazada pudiéndose iterar en ambos sentidos. Entre los diferentes tipos de operaciones que proporcionan las listas destacan:

       Inserciones y borrados en tiempo constante al principio y al final.

       Inserciones y borrados en tiempo constante en posiciones intermedias.

**Operaciones para el tipo de dato list**

|  |  |  |
| --- | --- | --- |
| *Constructores y asignación* |  |  |
| list<T> v | Constructor por defecto | *O(1)* |
| list<T> l (aList); | Constructor de copia | *O(n)* |
| l = aList | Asignación | *O(n)* |
| *Acceso a elementos* |  |  |
| l.front() | Primer valor de la colección | *O(1)* |
| l.back() | Último valor de la colección | *O(1)* |
| *Inserción y borrado* |  |  |
| l.push\_front (T) | Añade un elemento al principio de la lista | *O(1)* |
| l.push\_back (T) | Añade un elemento al final de la lista | *O(1)* |
| l.insert (iterator, T) | Inserta un nuevo elementos antes del iterador | *O(1)* |
| l.swap (list<T>) | Intercambia valores con otra lista | *O(1)* |
| l.pop\_front () | Borra el primer elemento de la lista | *O(1)* |
| l.pop\_back () | Borra el último elemento de la lista | *O(1)* |
| l.remove(T) | Eliminar todos los elementos iguales a uno dado | *O(n)* |

|  |  |  |
| --- | --- | --- |
| l.remove\_if(predicate) | Eliminar todos los valores que cumplan una condición | *O(n)* |
| l.erase (iterator) | Borra el elemento indicado por el iterador | *O(1)* |
| l.erase (iterator, iterator) | Borra un rango de valores | *O(1)* |
| *Tamaño* |  |  |
| l.size () | Número de elementos en la lista | *O(n)* |
| l.empty () | Cierto si la lista está vacía | *O(1)* |
| *Iteradores* |  |  |
| list<T>::iterator itr | Declara un nuevo iterador | *O(1)* |
| l.begin () | Iterador que referencia al primer elemento | *O(1)* |
| l.end () | Iterador que referencia al siguiente al último | *O(1)* |
| list<T>::reverse\_iterator ritr | Declara un nuevo reverse\_iterator |  |
| l.rbegin () | Reverse\_iterator que referencia al último elemento | *O(1)* |
| l.rend () | Reverse\_iterator que referencia al anterior al primero | *O(1)* |
| Otros métodos |  |  |
| l.reverse() | Invierte la lista | *O(n)* |
| l.sort() | Ordena los elementos de menor a mayor | *O(nlogn)* |
| l.merge(list<T>) | Mezcla con otra lista ordenada | *O(n)* |
| l.sort(comparision) | Ordena los elementos según una función | *O(nlogn)* |

***Map y Multimap***

Un map es una secuencia de pares (clave, valor) ordenados de menor a mayor en función de la clave. Las claves son únicas, esto es, cada clave tiene asociado un sólo valor. El tipo de iterador sobre esta clase de contenedores es bidireccional. Los componentes de un map son elementos de la clase pair<T1,T2>, que tiene las siguientes características:

**Operaciones para el tipo de dato pair**

|  |  |  |
| --- | --- | --- |
| *Constructores y asignación* |  |  |
| pair<T1,T2> p | Constructor por defecto | *O(1)* |
| pair<T1,T2> p (T1,T2) | Constructor con dos argumentos | *O(1)* |
| pair<T1,T2> p(aPair) | Constructor de copia | *O(1)* |
| *Área de datos (públicos)* |  |  |
| T1    first | Nombre de la variable donde se guarda el primer elemento del par |  |
| T2    second | Nombre de la variable donde se guarda el segundo elemento del par |  |
| Funciones | Nombre de la variable donde se guarda el segundo elemento del par |  |
| make\_pair (T1,T2) | Crea un par con los argumentos |  |

Hay que señalar que map es una secuencia de pares pair<const Key, Value>, y por lo tanto los iteradores para maps iteran sobre pares.

Un multimap es un map con la diferencia de que se permite la duplicidad de claves.

Para utilizar estas clases hay que incluir las librerías <map.h> y <multimap.h>.

**Operaciones para el tipo de dato map y multimap**

|  |  |  |
| --- | --- | --- |
| *Constructores y asignación* |  |  |
| map<K, V> m; | Constructor por defecto | *O(1)* |
| multimap<K, V> m | Constructor por defecto | *O(1)* |
| map<K, V> m (aMap) | Constructor de copia | *O(n)* |
| multimap<K, V>m (aMultimap) | Constructor de copia | *O(n)* |
| m = aMap | Asignación | *O(n)* |
| *Inserción y borrado* |  |  |
| m[key] (sólo para map) | Devuelve una referencia al valor asociado a key | *O(log n)* |
| m.insert(key\_value\_pair) | Inserta un par con clave y valor dados | *O(log n)* |
| m.erase(key) | Borra valor con clave asociada key | *O(log n)* |
| m.erase(iterator) | Borra el valor al que apunta un iterador | *O(log n)* |
| *Pruebas de inclusión* |  |  |
| m.size () | Número de elementos de la colección | *O(n)* |
| m.empty () | Cierto si la colección está vacía | *O(1)* |
| m.count(key) | Número de elementos con clave key | *O(log n)* |
| m.find(key) | Iterador sobre elemento con una clave dada | *O(log n)* |

|  |  |  |
| --- | --- | --- |
| m.lower\_bound(key)       (sólo para multimap) | Iterador sobre la primera ocurrencia de un elemento con clave key | *O(log n)* |
| m.upper\_bound(key)       (sólo para multimap) | Iterador sobre el siguiente elemento después del último con clave key | *O(log n)* |
| m.equal\_range(key)        (sólo para multimap) | Par de iteradores formado por lower\_bound y upper\_bound | *O(log n)* |
| *Iteradores* |  |  |
| map<K, V>::iterator itr | Declara un nuevo iterador | *O(1)* |
| m.begin () | Iterador que referencia al primer elemento | *O(1)* |
| m.end () | Iterador que referencia al siguiente al último | *O(1)* |
| map<K, V>::reverse\_iterator ritr | Declara un nuevo reverse\_iterator |  |
| m.rbegin () | Reverse\_iterator que referencia al último elemento | *O(1)* |
| m.rend () | Reverse\_iterator que referencia al anterior al primero | *O(1)* |

***Strings***

String es una clase que ofrece facilidades para el manejo de cadenas de caracteres. En C o C++, las cadenas están representadas por el tipo de dato *char \**y existen librerías como <string.h> que contienen funciones que facilitan su uso. En la STL este tipo de funciones están implementadas bajo la clase string (para utilizar strings se incluirá la librería <string>).

**Operaciones para el tipo de dato string**

|  |  |
| --- | --- |
| *Constructores* |  |
| string s | Constructor por defeto |
| string s ( “hola”) | Constructor con inicializador |
| string s (aString) | Constructor de copia |
| *Acceso a elementos* |  |
| s[i] | Acceso al elemento i-ésimo del string |
| s.substr(int pos,int len) | Subcadena que comienza en pos y tiene longitud len |
| s.c\_str() | Devuelve una cadena estilo C igual al string |
| *Inserción y borrado* |  |
| s.insert(int pos,string str) | Insetar antes de pos el string str |
| s.erase (int start, int len) | Eliminar desde s[start] hasta s[start+len] |
| s.replace(int start, int len,str) | Sustituir desde s[start] hasta s[start+len] por str |
| *Longitud* |  |
| s.length() | Longitud del string |
| s.resize(int,char) | Cambia el tamaño, rellenando con un valor |
| s.empty() | Cierto si el string es vacío |
| *Asignación* |  |
| s = s2 | Asignación de strings |
| s += s2 | Concatenación de strings |
| s + s2 | Nuevo string resultado de concatenar s y s2 |
| *Comparaciones* |  |
| s ==s2   s != s2 | Igualdad y desigualdad de strings |
| s < s2    s <= s2 | Comparaciones de strings (orden lexicográfico) |
| s > s2    s >= s2 | Comparaciones de strings (orden lexicográfico) |
| *Iteradores* |  |
| string::iterator itr | Declara un nuevo iterador |
| s.begin () | Iterador que referencia al primer elemento |
| s.end () | Iterador que referencia al siguiente al último |
| string::reverse\_iterator ritr | Declara un nuevo reverse\_iterator |
| s.rbegin () | Reverse\_iterator que referencia al último elemento |

|  |  |
| --- | --- |
| s.rend () | Reverse\_iterator que referencia al anterior al primero |
| *Operaciones de búsqueda* |  |
| s.find(string str, int pos) | Devuelve la posición en donde comienza la subcadena  str desde s[pos]. |
| s.find\_first\_of(str,pos) | Posición  en donde se encuentra el primer carácter que pertenece a  str desde s[pos]. |
| s.find\_first\_not\_of(str,pos) | Posición en donde se encuentra el primer carácter que no está en str desde s[pos]. |
| s.find\_last\_of(str,pos) | Posición  en donde se encuentra el último carácter que pertenece a  str desde s[pos]. |
| s.find\_las\_not\_of(str,pos) | Posición en donde se encuentra el último carácter que no está en str desde s[pos]. |
| *Operaciones E/S* |  |
| stream >> str | Entrada de strings |
| stream << str | Salida de strings |
| getline(stream,str,char) | Añade a str todos los caracteres de una línea de la entrada estándar hasta encontrar el carácter char. Por defecto char es igual a ‘\n’. |