STANDARD TEMPLATE LIBRARY

La **STL** es una de las más importantes características agregadas a C++ durante el proceso de estandarización. Suministra funciones, clases y estructuras de datos de propósitos generales basadas en distintos templates que facilitan la programación pudiéndose aplicar a casi cualquier tipo de datos. Por ejemplo, brinda el soporte necesario para el manejo de vectores, listas, colas, stacks, etc.

No es posible realizar un estudio detallado de todas las características de la STL debido a que es una librería muy grande. Una referencia completa se puede encontrar por ejemplo en la Parte 4 del libro C++: **The Complete Reference** de **Herbert Schildt**.

El núcleo de STL se basa en tres ítems fundamentales: algoritmos (**algorithms**), contenedores (**containers**) e iteradores (**iterators**). Su empleo en forma conjunta permite programar rápidamente soluciones a una gran variedad de problemas específicos.

Ventajas de su utilización

- Al ser estándar, se encuentra disponible en todos los compiladores y plataformas, permitiendo utilizarla en todos los proyectos, reduciendo notablemente el tiempo de desarrollo de las aplicaciones.
- Al utilizar una librería libre de errores se incrementa la robustez de la aplicación.
- Las aplicaciones se construyen a partir de algoritmos eficientes, quedando a cargo del programador la selección del algoritmo más rápido para una situación dada.
- Se incrementa la legibilidad del código, haciéndolo más fácil de mantener.
- Proporciona su propia gestión de memoria en forma automática y portátil, liberando al usuario de problemas tales como las limitaciones del modelo de memoria de la PC.

Palabra clave **typename**

Además de ser un sustituto para la palabra class en la declaración de un template, **typename** es un especificador de tipo. Indica que el identificador que se encuentra a continuación es un tipo, aunque no se haya definido todavía. Es una declaración adelantada para que el compilador no rechace al identificador. Por ejemplo en la función:

```
template <class T> void vPrintVec(vector<T> v, bool vert)
{
    typename vector<T>::iterator ForIter;
   // otras sentencias
}
```

si no se utiliza typename el compilador genera un error “tipo no definido” en esa línea.

Algorithms

La STL proporciona una amplia variedad de algoritmos comunes entre los que se incluyen los de ordenación, búsqueda y numéricos. Se pueden utilizar con vectores, estructuras de datos de la librería y las definidas por el usuario. Se comportan como operadores que actúan sobre sus elementos. Se comportan como funciones genéricas que obtienen la información necesaria para su funcionamiento a partir del análisis de los argumentos con que son invocadas.

Containers

Los contenedores son clases genéricas que pueden incluir objetos (estructuras de datos). Por ejemplo la clase **vector** define un arreglo dinámico, **deque** (pronunciado *deck* es un acrónimo de **d**ouble-**e**nded **que**ue), que puede expandirse o contraerse dinámicamente tanto al inicio como al final y **list** que provee una lista lineal. Estos son llamados ***contenedores secuenciales*** porque en la terminología de STL una secuencia es una lista lineal. También se definen ***contenedores asociativos*** que almacenan sus miembros en forma de árbol indexados que resultan adecuados para realizar accesos aleatorios mediante claves (keys). Por ejemplo un **map** almacena pares clave/valor accediendo a los valores por medio de su clave (valor único). Los contenedores asociativos mantienen sus elementos ordenados.

Cada clase contenedora define su propio conjunto de funciones; por ejemplo una lista incluye funciones para insertar, borrar y combinar elementos, mientras que una pila (**stack**) incluye funciones para agregar (**push**) o sacar (**pop**) valores. Se puede cambiar de **tipo** de contenedor utilizando los ***adaptadores*** que cambian su interfaz (métodos públicos) para incorporar un conjunto de operaciones diferente. En la mayoría de los casos, el nuevo contenedor requiere únicamente de un subconjunto de las capacidades que proporciona el contenedor original.

Iterators

Teniendo en cuenta que los contenedores son clases genéricas, y los algoritmos que operan sobre ellos son funciones genéricas, se tuvo que desarrollar el concepto de iterador como elemento o nexo de conexión entre ambos. Los algoritmos aceptan iteradores como argumentos. Estos objetos actúan como punteros (se pueden incrementar, decrementar y aplicar el operador *), posibilitando el acceso al contenido de las distintas estructuras de datos del mismo modo que usando el índice en un array. Existen cinco tipos de iteradores:

- Random Access: almacena y recupera valores. Los elementos se acceden en forma aleatoria.
- Bidirectional: Almacena y recupera valores. Se puede acceder hacia adelante o hacia atrás.
- Forward: almacena y recupera valores. Se puede acceder sólo elementos hacia el final.
- Input: sólo recupera valores. Movimientos forward.
- Output: sólo almacena valores. Movimientos forward.

Los iteradores se declaran mediante el tipo **iterator** que se encuentra definido en varios contenedores. STL soporta los iteradores en sentido inverso (reverse iterators). Si se tiene un interador inverso apuntando al final de una secuencia, un incremento del mismo hará que se apunte a un elemento anterior.

En la bibliografía se utilizan términos abreviados para los distintos tipos de iteradores utilizados en un template:

Término	Representa un iterador
BiIter	Bidirectional.
ForIter	Forward.
InIter	Input.
OutIter	Outer.
RandIter	Random access.

Otros elementos de la STL

La librería utiliza también otros componentes estándar: de asignación de memoria (allocators), predicates, y funciones objeto.

allocator: son objetos diseñados con el propósito de hacer que la STL sea flexible e independiente del modelo de memoria que forme parte del hardware, facilitando al programador el manejo de punteros y referencias. Personalizados. Cada contenedor tiene definido su allocator, que maneja la memoria necesaria. Aunque generalmente no es necesario, el usuario puede definir su propio allocator.

predicate: es un tipo de función que retorna **bool** (true/false) o una instancia de un objeto con una función miembro *bool operator()*. Las condiciones para determinar cuándo retornar true o false son puestas por el usuario. Pueden ser unarios (un argumento) o binarios (dos argumentos). Cuando se requieran funciones predicate unary se recurre al tipo **UnPred**. Si se necesita un predicate binary se usa **BinPred**. Para ambos los parámetros deben ser del mismo tipo que el almacenado en el contenedor. Las funciones de comparación se escriben con el tipo **Comp**. Casi todos los **algoritmos** de la STL utilizan predicate como último argumento.

funciones objeto: varios de los algoritmos proporcionados por la STL permiten recurrir a una una función que adapte su funcionalidad. Las funciones objeto son una generalización del puntero a función de C.

Archivos de cabecera

Además de los headers requeridos por las diversas clases de la STL, la librería estándar de C++ incluye los archivos <utility> y <functional>. Por ejemplo el template **pair**, diseñado para contener un par de valores (usado por **map**), se define en utility. Los template declarados en functional ayudan en la construcción de objetos que definen **operator()**, que puede ser usada en lugar de los punteros a function. Existen varias funciones objeto predefinidas que serán analizadas más adelante.

Clases contenedoras

Se listan a continuación los contenedores que se encuentran definidos, y los archivos de cabecera necesarios para su uso.

Secuenciales

vector, <vector>: puede cambiar el tamaño del array dinámicamente. Inserción y eliminación rápida en la parte final. Acceso rápido a cualquier elemento, soporta el operador índice.

list, <list>: secuencia lineal no ordenada. Suelen ser implementados como listas doblemente enlazadas. Dispone de mecanismos eficientes para insertar elementos en cualquier posición (el tiempo no depende el número de elementos almacenados). Los elementos no pueden ser accedidos por subíndices como en un vector, por lo que las operaciones de acceso utilizan tiempos proporcionales al número de elementos.

deque, <queue>: cola de doble terminación. Comparte las características de las colas (ingresa los nuevos elementos por un extremo, pudiendo los anteriores ser extraídos por el extremo opuesto) y de las pilas (el último elemento introducido por el principio puede ser extraído por el mismo extremo). Suelen ser implementados como matrices bidimensionales. Inserciones y eliminaciones rápidas en la parte inicial o final. Acceso aleatorio a cualquier elemento.

string, <string>: contenedor adaptado a operaciones con cadenas de caracteres.

bitset, <bitset>: simula un array de elementos booleanos, optimizado para minimizar el espacio de memoria. No existen iteradores para recorrerlos; sus elementos se acceden utilizando el operador subíndice [ ]. Dispone de varias funciones para realizar operaciones a nivel de bits.

Asociativos

map, <map>: almacena pares clave/valor, cada clave se asocia a un **único valor**. El key–value puede ser de cualquier tipo, a condición de que sus elementos puedan ser ordenados según un criterio (por defecto el operador <). Dispone de mecanismos de inserción y borrado muy eficientes y no existe límite de tamaño. La estructura crece o disminuye en forma automática. Permite el uso del operador subíndice para los elementos de la clave. Se denominan también diccionarios, tablas y matrices asociativas.

multimap, <map>: a diferencia del map, cada clave puede estar asociada a **dos o más valores** (permite claves duplicadas).

set, <set>: es un conjunto donde cada elemento es único. Dispone de mecanismos eficientes para inclusión, inserción y eliminación de elementos, y soporta claves únicas. Las búsquedas son rápidas.

multiset, <set>: igual comportamiento que set pero permite la existencia de claves duplicadas.

Adaptadores

stack, <stack>: elementos tipo pila LIFO (Last Input First Output) que permite inserciones y eliminaciones sólo en la parte superior.

queue, <queue>: se comporta como una cola FIFO (First Input First Output).

priority_queue, <queue>: es una cola prioritaria. El elemento de mayor prioridad siempre es el primero en salir.

Dado que los nombres de los tipos de datos genéricos (T, *placeholders*) en la declaración de la plantilla de una clase son arbitrarios, las clases contendoras declaran mediante un typedef versiones de nombres más concretas:

size_type	algún tipo de entero.
reference	una referencia a un elemento.
const_reference	una referencia **const** a un elemento.
iterator	un iterador.
const_iterator	un iterador constante.
reverse_iterator	un iterador inverso
const_reverse_iterator	un const iterador inverso.
value_type	el tipo de un valor almacenado en un contenedor.
allocator_type	el tipo de un allocator.
key_type	el tipo de una clave.
key_compare	el tipo de una función que compara dos claves.
value_compare	el tipo de una función que compara dos valores.

El funcionamiento interno de la STL es bastante sofisticado, pero su utilización es bastante simple. La primera decisión es determinar la clase de container que se necesita. Salvo el caso de bitset, un contenedor crecerá (decrecerá) automáticamente cuando se agreguen (eliminen) elementos. Un **vector** es una buena opción para arreglos de acceso aleatorio y no se necesita insertar ó eliminar demasiados elementos. Una **list** realiza mejor las operaciones estas operaciones a expensas de velocidad. Si se necesita una búsqueda asociativa debe utilizarse un **map** que tiene una sobrecarga adicional de tiempos.

Las funciones a utilizar para agregar, eliminar o acceder a los elementos dependen del contenedor seleccionado, siendo las más utilizadas **insert**(), **erase**(), **push**/**pop_back**() y **push**/**pop_front**(). Las secuencias y los contenedores asociativos cuentan con las funciones **begin**() y **end**() que retornan iteradores que pueden utilizarse para recorrer los datos. En el caso asociativo se provee la función **find**() para localizar un elemento mediante su key.

Los algoritmos permiten, además de cambiar el contenido de un contenedor de alguna manera determinada, transformar un tipo de secuencia en otro.

vector

Es el contenedor que se adapta a la mayoría de los requerimientos. La clase maneja un arreglo dinámico que puede crecer según la necesidad. El acceso a los elementos se puede realizar también mediante su índice. Su template es de la forma:

template < **class** **T**, **class** **Allocator** = allocator<**T**> > **class** **vector**

donde **T** es el tipo de datos y **Allocator** especifica el allocator que por defecto es el estándar.

La clase cuenta con los siguientes constructores:

- **explicit** **vector**(**const** **Allocator** &*a* = Allocator( ));
- **explicit** **vector**(**size_type** *num*, **const** **T** &*val* = T(), **const** **Allocator** &*a* = Allocator( ));
- **vector**(**const** **vector**<**T**, **Allocator**> &*ob*);
- **template** <**class** **InIter**> **vector**(**InIter** *start*, **InIter** *end*, **const** **Allocator** &*a* = Allocator( ));

El modificador **explicit** evita que se hagan conversiones implícitas (automáticas) entre tipos de parámetros (si la función espera un float no se le puede pasar un int). La primera forma se utiliza para crear un vector vacío. La segunda opción construye un vector que tiene *num* elementos con el valor *val* (que puede tener un valor por defecto). La tercera forma construye un vector con los mismos elementos que *ob*. Con la última forma se obtiene un vector que contiene los elementos en el rango especificado por los iteradores *start* y *end*. Por ejemplo:

```
vector<int> iv;              // crea un ventor de enteros con longitud cero
vector<int> iv2(iv);         // crea un vector de enteros a partir del anterior
vector<char> cv(5);          // crea un vector de caracteres con 5 elementos
vector<char> cv(5,  'x');    // vector de 5 caracteres inicializados con ‘x’
```

Cualquier objeto que sea almacenado en un vector debe definir un constructor por defecto y los operadores < y == (al menos). Todos los tipos básicos satisfacen estas condiciones y tienen definidos los operadores ==, <, <=, !=, >, >= y [].

En la siguiente tabla se muestran algunas de las funciones miembro de la clase (se destacan las más utilizadas):

Método	**Descripción**
reference back(); const_reference back() const;	Retorna una referencia al último elemento en el vector.
iterator **begin**(): const_iterator begin() const;	Retorna un iterador al primer elemento del vector.
void clear();	Elimina todos los elementos del vector.
bool empty() const;	Retorna true si el vector que lo invoca está vacío.
iterator **end**(); const_iterator end() const;	Retorna un puntero a la posición inmediata después del último elemento del vector.
iterator **erase**(iterator *i*);	Elimina el elemento apuntado por *i*. Retorna un iterador al elemento posterior al eliminado.

iterator **erase**(iterator *f*, iterator *l*);	Elimina los elementos en el rango *f* a *l*. Retorna un iterador al elemento siguiente al último eliminado.
reference front(); const_reference front() const;	Retorna una referencia al primer elemento.
iterator **insert**(iterator *i*, const T &*val*);	Inserta *val* inmediatamente antes del elemento *i*. Retorna un iterador apuntando a ese elemento.
template <class InIter> void **insert** (iterator *i*, InInter *start*, InIter *end*);	Inserta la secuencia definida por *start* y *end* inmediatamente antes del elemento *i*.
reference operator[](size_type *i*) const; const_reference oprtr[](size_t *i*) const;	Retorna una referencia al elemento *i*.
reference& **at**(size_type *i*) const; const_reference& at(size_t *i*) const;	Retorna una referencia al elemento *i*. Es más segura que [] porque controla los límites de acceso. Si *i* está fuera del rango permitido lanza una excepción.
void **pop_back**();	Remueve el último elemento en el vector.
void **push_back**(const T &*val*);	Agrega al final un elemento con el valor *val*.
size_type **size**() const;	Retorna el número de elementos actual del vector.

```
/***** Ejemplo utilización clase vector
#include <iostream>
#include <vector>
#include <windows.h>

#define     VERT     1
#define     HORI     0

using namespace std;

template <class T> void vPrintVec(vector<T> v, bool vert)
{
   typename vector<T>::iterator ForIter;

   ForIter = v.begin();                    // puntero al primer elemento
   cout << endl;
   while (ForIter != v.end() )             // mientras no sea el último elemento
   {
      cout << *ForIter++ << " ";           // aumento el iterador
      if (vert)
         cout << endl;
   }
   cout << "\n\n";
}

int main()
{
   SetConsoleOutputCP(1252);
   SetConsoleCP(1252);
```

```
vector<char> cVec(10);                  // crea un vector de 10 caracteres
unsigned int i,
             lim = cVec.size();         // tamaño actual del vector

cout << "VECTOR DE CARACTERES\n\nTamaño original = " << lim << endl;
for(i=0; i<10; i++)
{
   cVec[i] = i + 'a';                   // asigna valores consecutivos
}
vPrintVec(cVec, HORI);

for(i=0; i<10; i++)
{
   cVec.push_back(i  + 'A');            // agrego valores al final, crece solo
}
cout << "Nuevo tamaño = " << cVec.size() << endl;
vPrintVec(cVec, HORI);

cVec.pop_back();                        // extrae el último elemento
vector<char>::iterator ForIter = cVec.begin();
ForIter += 2;                           // apunta al 3er. elemento
cVec.insert(ForIter, 3, 'X');           // inserta 3 X en las posiciones 2, 3 y 4
cout << "Después de la insertar 3 'X' (en la tercera posición): ";
vPrintVec(cVec, HORI);
cout << "Tamaño después de la inserción = " << cVec.size() << endl;

char car=cVec.front();                  // otra forma de obtener el primer elemento
cout << "\nPrimer elemento: " << car;
car=cVec.back();                        // otra forma de obtener el último elemento
cout << ", último elemento: " << car << endl;

cout << "\n\n\nVECTOR DE FLOTANTES\n";
float f;
vector<float> fVec;                     // crea un vector de flotantes
for(i=0; i<7; i++)                      // no utilizar [i] porque está vacío
{
   fVec.push_back(float(i * 3.1415927));    // agrego valores al final, crece solo
}
vPrintVec(fVec, VERT);

f = *(fVec.end()-1);                    // accedo al último elemento
cout << "Último elemento  = " << f << endl;
size_t tam = fVec.size();
f = fVec[tam-2];                        // accedo al penúltimo elemento con []
cout << "Penúltimo elemento  = " << f << endl;
f = fVec.at(tam-3);                     // accedo al antepenúltimo elemento con .at()
cout << "Antepenúltimo elemento  = " << f << endl;

vector<float>::iterator fForIter;
fForIter = fVec.begin();                // iterador para flotantes

fVec.insert(fForIter+2, fForIter, fForIter+4);
cout << "\nSe inserta en la tercera posición las componentes 1º, 2º, 3º y 4º";
vPrintVec(fVec, HORI);

fForIter+= 2;                           // tercer elemento
fVec.erase(fForIter, fForIter+3);       // elimino componentes 3º, 4º y 5º
cout << "\nSe eliminaron las componentes 3º, 4º y 5º";
vPrintVec(fVec, HORI);
```

```
   return 0;
}
fin del ejemplo *****/


/***** Ejemplo vector de complejos
#include <iostream>
#include <vector>
#include "complejos.h"

using namespace std;

int main()
{
   complejo c1(1.1, 2.2),      // se crean e inicializan dos complejos (constructor general)
            c2(3.3, 4.4),      // se crea un complejo con el constructor por defecto
            c3,
            c4(7.5),           // se crea un complejo con el valor por defecto (0.0) del 2do. argumento
            suma = c1 + c2,    // se crea un complejo a partir del resultado (constructor de copia)
            resta, producto, cociente;   // inicializados por el constructor por defecto

   c3.setReal(5.0);            // se fijan los valores de las partes real e imaginaria de c3
   c3.setImag(2.0);

   resta = c1 - c2;
   producto = c1 * c2;
   cociente = c1 / c2;

   cout << "Primeros cuatro valores = " << c1 << ", " << c2 << ", " << c3 << ", " << c4 << endl << endl;
   cout << "Suma:     " << suma << endl << endl;
   cout << "Resta:    " << resta << endl << endl;
   cout << "Producto: " << producto << endl << endl;
   cout << "Cociente: " << cociente << endl << endl;

   vector<complejo> cVec(4);      // crea un vector de 4 complejos
   unsigned int i;

   cVec[0] = c1;                  // asigna valores consecutivos
   cVec[1] = c2;
   cVec[2] = c3;
   cVec[3] = c4;
   cout << "\nVECTOR DE COMPLEJOS\n\nTamano original = " << cVec.size() << endl;

   cVec.push_back(suma);          // agrego valores al final, crece solo
   cVec.push_back(resta);
   cVec.push_back(producto);
   cVec.push_back(cociente);
   cout << "\nTamano aumentado = " << cVec.size() << endl << endl;

   for(i=0; i<cVec.size(); i++)
   {
      cout << "Valor en la posicion " << i << " = " << cVec[i] << endl;
   }

   vector<complejo>::iterator ForIter;
   ForIter = cVec.begin();        // iterador para complejos
   ForIter+= 2;                   // tercer elemento
   cVec.erase(ForIter, ForIter+2);  // elimino componentes 3ra y 4ta
   cVec.push_back(c1);            // agrego valores al final, crece solo
```

```
    cout << "\nSe eliminaron las componentes 3ra, y 4ta y se agrego la 1ra al final\n";
    for(i=0; i<cVec.size(); i++)
    {
       cout << "Valor en la posicion " << i << " = " << cVec[i] << endl;
    }
    cout << endl << endl;

    return 0;
}
fin del ejemplo *****/
```

list

Esta clase maneja una lista lineal bidireccional. A diferencia de un vector, los elementos sólo pueden ser accedidos secuencialmente. Al ser bidireccional el acceso se puede realizar desde el comienzo al final ó desde el final al comienzo. Su template es de la forma:

template < **class T**, **class Allocator** = allocator<**T**> > **class list**

donde **T** es el tipo de datos y **Allocator** especifica el allocator que por defecto es el estándar.

La clase cuenta con los siguientes constructores:

- **explicit list**(**const Allocator** *&a* = Allocator( ));
- **explicit list**(**size_type** *num*, **const T** *&val* = T(), **const Allocator** *&a* = Allocator( ));
- **list**(**const list**<**T**, **Allocator**> *&ob*);
- **template** <**class InIter**> **list**(**InIter** *start*, **InIter** *end*, **const Allocator** *&a* = Allocator( ));

La primera forma se utiliza para crear una lista vacía. La segunda opción construye una lista que tiene *num* elementos con el valor *val* (que puede tener un valor por defecto). La tercera forma construye una lista con los mismos elementos que *ob*. Con la última forma se obtiene una lista que contiene los elementos en el rango especificado por los iteradores *start* y *end*.

Si se utiliza con tipos de datos definidos por el usuario, los mismos deben proporcionar un constructor por defecto y definir los operadores de comparación que sean necesarios. Para los tipos de datos básicos se encuentran definidos los operadores **==**, **<**, **<=**, **!=**, **>** y **>=**.

En la siguiente tabla se muestran algunas de las funciones miembro de la clase (se destacan las más utilizadas):

Método	**Descripción**
reference back(); const_reference back() const;	Retorna una referencia al último elemento en la lista.
iterator **begin**(): const_iterator begin() const;	Retorna un iterador al primer elemento.

void clear();	Elimina todos los elementos de la lista.
bool empty() const;	Retorna true si la lista que lo invoca está vacía.
iterator **end**(); const_iterator end() const;	Retorna un puntero a la posición inmediata después del último elemento de la lista.
iterator **erase**(iterator *i*);	Elimina el elemento apuntado por *i*. Retorna un iterador al elemento posterior al eliminado.
iterator **erase**(iterator *f*, iterator *l*);	Elimina los elementos en el rango *f* a *l*. Retorna un iterador al elemento siguiente al último eliminado.
reference front(); const_reference front() const;	Retorna una referencia al primer elemento.
iterator **insert**(iterator *i*, const T &*val*);	Inserta *val* inmediatamente antes del elemento *i*. Retorna un iterador apuntando a ese elemento.
void **insert**(iterator *i*, size_type *num*, const T &*val*)	Inserta *num* copias de *val* inmediatamente antes del elemento especificado por *i*.
template <class InIter> void **insert** (iterator *i*, InInter *start*, InIter *end*);	Inserta la secuencia definida por *start* y *end* inmediatamente antes del elemento *i*.
void **merge**(list<T, Allocator> &*ob*); template <class Comp> void merge(<list<T, Allocator> &*ob*, Comp *cmpfn*);	Combina la lista contenida en *ob* con la que invoca el método. Al finalizar *ob* queda vacía. En la segunda forma, se puede especificar una función de comparación para determinar cuando un elemento es menor que otro.
void **pop_front**( );	Remueve el primer elemento en la lista.
void **pop_back**();	Remueve el último elemento en la lista.
void **push_front**(const T &*val*);	Agrega al comienzo un elemento con el valor *val*.
void **push_back**(const T &*val*);	Agrega al final un elemento con el valor *val*.
void **remove**(const T &*val*);	Elimina los elementos con el valor *val* de la lista.
void **reverse**();	Invierte la lista que invoca al método.
size_type **size**() const;	Retorna el número de elementos actual del vector.
void **sort**(); template <class Comp> void **sort** (Comp *cmpfn*);	Ordena la lista. En la segunda forma, se puede especificar una función de comparación para determinar cuando un elemento es menor que otro.

void splice(iterator *i*, list <T, Allocator> &*obj*);	Inserta el contenido de *obj* en la posición *i*. Luego de la operación *obj* queda vacía.
void splice(iterator *i*, list <T, Allocator> &*obj*, iterator *el*);	Elimina el elemento apuntado por *el* de *obj* y lo inserta en la posición *i*.
void splice(iterator *i*, list <T, Allocator> &*obj*, iterator *start*, iterator *end*);	Elimina el rango de elementos entre start y end de *obj* y lo inserta a partir de la posición *i*.

```
/***** Ejemplo utilización clase list
#include <iostream>
#include <list>
#include <stdlib.h>

using namespace std;

template <class T> void vPrintLst(list<T> lst)
{
   typename list<T>::iterator ForIter;

   ForIter = lst.begin();                  // puntero al primer elemento
   cout << endl;
   while (ForIter != lst.end() )           // mientras no sea el último elemento
   {
      cout << *ForIter++ << " ";           // aumento el iterador
   }
   cout << "\n\n";
}

int main()
{
   list<int> lst;                          // crea una lista vacia
   list<int>::iterator ForIter,            // crea un iterador
                       stIter, enIter;     // iteradores de inicio y final
   unsigned int i;
   int valor;

   for(i=0; i<10; i++)
   {
      valor = rand();
      lst.push_back(valor);                // agrega un elemento al final
      lst.push_front(valor);               // agrega un elemento al principio
   }
   cout << "LISTA DE ENTEROS\n\nTamano inicial = " << lst.size();
   vPrintLst(lst);

   stIter = enIter = ForIter = lst.begin();    // iteradores desde el comienzo
   advance(stIter, 2);                         // tercer elemento
   advance(enIter, 12);                        // decimotercer elemento
   lst.erase(stIter, enIter);                  // elimina elementos
   cout << "\nSe eliminan los elementos desde el tercero al decimotercero:";
   vPrintLst(lst);

   lst.sort();
   cout << "\nSe ordenaron los elementos de la lista (menor a mayor): ";
   vPrintLst(lst);
   valor=lst.front();                    // otra forma de obtener el primer elemento
```

```
   cout << "Primer elemento: " << valor;
   valor=lst.back();                    // otra forma de obtener el Ãºltimo elemento
   cout << ", ultimo elemento: " << valor << endl;

   lst.reverse();
   cout << "\nSe ordenaron los elementos de la lista (mayor a menor): ";
   vPrintLst(lst);

   return 0;
}
```

fin del ejemplo *****/

/***** Ejemplo **sort list** con números complejos

```
#include <iostream>
#include <list>
#include "complejos.h"
#include <math.h>

using namespace std;

template <class T> void vPrintLst(list<T> lst)
{
   typename list<T>::iterator ForIter;

   ForIter = lst.begin();                  // puntero al primer elemento
   cout << endl;
   while (ForIter != lst.end() )           // mientras no sea el último elemento
   {
      cout << *ForIter++ << endl;          // aumento el iterador
   }
   cout << "\n\n";
}

bool cmpfn (complejo &c1, complejo &c2)
{
   float fMod1, fMod2;

   fMod1 = sqrt(pow(c1.getReal(), 2) + pow(c1.getImag(), 2));
   fMod2 = sqrt(pow(c2.getReal(), 2) + pow(c2.getImag(), 2));

   return (fMod1 < fMod2);
}

int main()
{
   complejo c1(1.1, 2.2),          // se crean e inicializan dos complejos (constructor general)
            c2(3.3, 4.4),          // se crea un complejo con el constructor por defecto
            c3,
            c4(7.5),               // se crea un complejo con el valor por defecto (0.0) del 2do. argumento
            suma = c1 + c2,        // se crea un complejo a partir del resultado (constructor de copia)
            resta, producto, cociente;    // inicializados por el constructor por defecto

   c3.setReal(5.0);                // se fijan los valores de las partes real e imaginaria de c3
   c3.setImag(2.0);

   resta = c1 - c2;
   producto = c1 * c2;
   cociente = c1 / c2;
```

```
    list<complejo> lst;              // crea una lista vacia
    lst.push_back(suma);             // agrego valores al final, crece sola
    lst.push_back(resta);
    lst.push_back(producto);
    lst.push_back(cociente);
    lst.push_front(c1);              // agrego valores al inicio, crece sola
    lst.push_front(c2);
    lst.push_front(c3);
    lst.push_front(c4);

    cout << "LISTA DE COMPLEJOS\n\nTamano inicial = " << lst.size();
    vPrintLst(lst);

    lst.sort(cmpfn);
    cout << "\nComplejos ordenados por modulo (menor a mayor): ";
    vPrintLst(lst);

    return 0;
}
```

fin del ejemplo *****/

/***** Ejemplo de **list** usando **merge** (combinación)

```
#include <iostream>
#include <list>

using namespace std;

template <class T> void vPrintLst(list<T> lst)
{
    typename list<T>::iterator ForIter;

    ForIter = lst.begin();                  // puntero al primer elemento
    cout << endl;
    while (ForIter != lst.end() )           // mientras no sea el Ãºltimo elemento
    {
        cout << *ForIter++ << "  ";         // aumento el iterador
    }
    cout << "\n\n";
}

int main()
{
    list<int> lst1, lst2;
    int i;

    for(i=0; i<10; i++)
    {
        lst1.push_back(i);
    }
    for(i=0; i<10; i++)
    {
        lst2.push_front(i);
    }
    cout << "LISTA DE ENTEROS\n\nLista 1 usando push_back:";
    vPrintLst(lst1);
    cout << "Lista 2 usando push_front:";
    vPrintLst(lst2);

    lst1.merge(lst2);
```

```
    cout << "\nResultado de merge:";
    vPrintLst(lst1);
    if(lst2.empty())
    {
       cout << "lst2 quedo vacia\n\n";
    }

    return 0;
}
fin del ejemplo *****/
```

map

Esta clase maneja un contenedor asociativo con una clave única para cada elemento. Los valores son accedidos por medio de su clave. Su template es de la forma:

```
template < class Key, class T, class Comp = less<Key>,
           class Allocator = allocator<T> > class map
```

donde **K** es el tipo de datos de la clave, **T** es el tipo de datos de los valores almacenados (mapped), **Comp** es una función que compara dos claves y **Allocator** especifica el allocator que por defecto es el estándar.

La clase cuenta con los siguientes constructores:

- **explicit map**(**const Comp** &cmpfn = Comp( ), **const Allocator** &*a* = Allocator( ));
- **map**(**const map**<*Key*, *T*, *Comp*, *Allocator*> &*ob*);
- **template** <**class InIter**> **map**(**InIter** *start*, **InIter** *end*,
  const Comp &*cmpfn* = Comp( ) **const Allocator** &*a* = Allocator( ));

La primera forma se utiliza para crear un map vacío. La segunda opción construye un map que con los mismos elementos que *ob*. Con la última forma se obtiene un map que contiene los elementos en el rango especificado por los iteradores *start* y *end*. La función *cmpfn*, si se especifica, determina el orden del map.

Si se utiliza con tipos de datos definidos por el usuario, los mismos deben proporcionar un constructor por defecto y definir los operadores de comparación que sean necesarios. Para los tipos de datos básicos se encuentran definidos los operadores ==, <, <=, !=, > y >=.

En la siguiente tabla se muestran algunas de las funciones miembro de la clase (se destacan las más utilizadas):

Método	**Descripción**
iterator **begin**(): const_iterator begin() const;	Retorna un iterador al primer elemento.
void clear();	Elimina todos los elementos del map.

size_type **count**(const key_type &*k*) const;	Retorna el número de veces que aparece k en el map (0 ó 1, no se permiten claves duplicadas).
bool empty() const;	Retorna true si el map que lo invoca está vacío.
iterator **end**(); const_iterator end() const;	Retorna un puntero a la posición inmediata después del último elemento.
iterator **erase**(iterator *i*);	Elimina el elemento apuntado por *i*.
iterator **erase**(iterator *f*, iterator *l*);	Elimina los elementos en el rango *f* a *l*.
size_type **erase**(const key_type &*k*)	Elimina el elemento cuya clave tiene el valor *k*.
iterator **find**(const key_type &*k*); const_iterator find(const key_type &*k*) const;;	Retorna un iterador a la clave *k*. Si no se encuentra se retorna un iterador al final del map.
iterator **insert**(pair<K,T>(*key*, *val*);	Inserta el par *key* , *val* en el map.
template <class InIter> void **insert** (InInter *start*, InIter *end*);	Inserta un rango de valores.
pair<iterator, bool> insert(const value_type &*val*);	Inserta *val* en el map. Retorna par <iterator, bool>. bool = true si el elemento no existía (se inserta).
reference **operator[ ]** (const key_type &*i*)	Retorna una referencia al elemento especificado por *i*. Si no existe se lo inserta.
size_type **size**() const;	Retorna el número de elementos actual del map.

Los pares Key/value (clave/valor) son almacenados en el map como objetos del tipo **pair**:

```
template < class Ktype, class Vtype > struct pair
{
    typedef Ktype first_type;          // tipo de la clave.
    typedef Vtype second_type;         // tipo del valor.
    Ktype first;                       // contiene la clave.
    Vtype second;                      // contiene el valor.
}
```

Se puede construir un par utilizando el constructor o con la función **make_pair**():

```
template <class Ktype, class Vtype> pair<Ktype, Vtype>
                        make_pair(const Ktype &k, const Vtype &v)
```

Los tipos de los objetos son determinados automáticamente por el compilador.

```
/***** Ejemplo de map, tabla de código ASCII
#include <iostream>
```

```
#include <list>
#include <iostream>
#include <map>

using namespace std;

template <class K, class V> void vPrintMp(map<K, V> mp)
{
   typename map<K, V>::iterator ForIter;

   ForIter = mp.begin();                        // puntero al primer elemento
   cout << endl;
   while (ForIter != mp.end() )                 // mientras no sea el Ãºltimo elemento
   {
      cout << ForIter->first << "  " << ForIter->second << endl;
      ForIter++;                                // aumento el iterador
   }
   cout << "\n\n";
}

int main()
{
   map<char, int> cm;
   map<char, int>::iterator ptr, ptr2;
   int i;
   char ch;

   for(i=0; i<26; i++)
   {
      cm.insert(pair<char, int>('A'+i, 65+i));
   }
   cout << "MAP DE CARACTERES\n\n:";
   vPrintMp(cm);

   cout << "Elimino la mitad del map:\n";
   ptr = ptr2 = cm.begin();
   advance(ptr2, int(cm.size()/2));
   cm.erase(ptr, ptr2);
   vPrintMp(cm);

   cout << "Ingrese una clave: ";
   cin >> ch;
   ptr = cm.find(ch);
   if(ptr != cm.end())
   {
      cout << "Su codigo ASCII es " << ptr->second << endl;
   }
   else
   {
      cout << "Clave no encontrada\n";
   }
   cout << "Ingrese una clave: ";
   cin >> ch;
   ptr = cm.find(ch);
   if(ptr != cm.end())
   {
      cout << "Su codigo ASCII es " << ptr->second << endl;
   }
   else
   {
```

```
        cout << "Clave no encontrada\n";
    }

    i = cm['P'];
    cout << "Acceso mediante el operador []: el codigo de 'P' es = " << i << endl << endl;
    cout << "Si no existe lo inserta, el codigo de 'b' es = " << cm['b'] << endl << endl;
    cm['b'] = 125;
    vPrintMp(cm);

    return 0;
}
fin del ejemplo *****/


/***** Ejemplo de map con persona, la clave es el número de documento

#include "persona.h"
#include <map>

using namespace std;

template <class K, class V> void vPrintMp(map<K, V> mp)
{
    typename map<K, V>::iterator ForIter;

    ForIter = mp.begin();                       // puntero al primer elemento
    cout << endl;
    while (ForIter != mp.end() )                // mientras no sea el último elemento
    {
        cout << ForIter->first << "  " << ForIter->second << endl;
        ForIter++;                              // aumento el iterador
    }
    cout << "\n\n";
}

int main()
{
    map<unsigned, persona> persm;
    persona ped("Pedro", 1234, fecha(21, 10, 2000)),
            pab("Pablo", 1122, fecha(15, 1, 2005)),
            vil("Vilma", 2341, fecha(31, 01, 1999));

    persm.insert(pair<unsigned, persona>(ped.GetDocument(), ped));
    persm.insert(pair<unsigned, persona>(pab.GetDocument(), pab));
    persm.insert(pair<unsigned, persona>(vil.GetDocument(), vil));

    cout << "MAP DE PERSONAS:\n\n";
    vPrintMp(persm);

    cout << "La persona con documento 1122 es: \n" << persm.find(1122)->second << endl;
    cout << "Acceso mediante el operador []: \n" << persm[2341] << endl << endl;
    persm[1234] = pab;              // modifico los datos de la persona
    vPrintMp(persm);

    return 0;
}
fin del ejemplo *****/
```

stack

Esta clase es un adaptador de contenedores que le permite al programador obtener la funcionalidad de una pila LIFO (Last In First Out), donde sólo se provee un conjunto específico de funciones. Actúa como una envoltura (wrapper) para la clase original. maneja una lista lineal bidireccional. Su template es de la forma:

template < **class T**, **class Container** = deque<**T**> > **class stack**

donde **T** es el tipo de elementos almacenado, que debe ser el mismo que el de la clase contenedora original que debe ser secuencial (no funciona por ejemplo con map). Además la misma debe proveer los métodos **back**(), **push_back**() y **pop_back**(). Las clases vector, deque y list satisfacen estos requerimientos.

En la siguiente tabla se muestran algunas de las funciones miembro de la clase (se destacan las más utilizadas):

Método	**Descripción**
reference **top**();	Retorna el elemento al tope de la pila.
void **pop**();	Elimina el elemento del tope.
void **push**(const T *&val*);	Agrega al tope el elemento *val*.
size_type **size**();	Retorna la cantidad de elementos en la pila.
bool empty();	Retorna true si la pila está vacía.

```
/***** Ejemplo de stack con entero, vector de flotantes y deque de caracteres

#include <stack>
#include <vector>
#include <iostream>

using namespace std;

int main ()
{
   stack<int> entero;                        // vacio
   cout << "STACK DE ENTEROS\n\nTamano inicial: " << entero.size() << '\n';
   entero.push(5);                           // agrego elementos
   entero.push(-1);
   cout << "\nSe agregan 2 valores, tamano: " << entero.size() << '\n';
   int valor = entero.top();                 // obtengo el valor del tope
   cout << "\tUltimo valor ingresado: " << valor << '\n';
   entero.pop();                             // elimino ese valor
   cout << "\nSe elimina un valor, tamano: " << entero.size() << '\n';
   valor = entero.top();                     // nuevo valor del tope
   cout << "\tValor en el tope: " << valor << "\n\n";

   vector<float> myvector (2, 5.5);          // vector con 2 elementos = 5.5
   myvector[1] = myvector[1] + 7.5;          // segundo valor = 5.5 + 7.5
   stack<float, vector<float> > stvect(myvector);   // stack de vector inicializado
   cout << "\nSTACK DE VECTOR\n\nTamano inicial: " << stvect.size() << '\n';
```

```
    float fvalor = stvect.top();
    cout << "\tValor en el tope: " << fvalor << '\n';
    stvect.pop();
    fvalor = stvect.top();
    cout << "Se elimina un valor\n\tValor en el tope: " << fvalor << '\n';

    deque<char> mydeque;                    // cola doble vacia
    mydeque.push_back('A');
    mydeque.push_front('H');
    mydeque.push_back('M');
    mydeque.push_front('Z');                // Z H A M
    stack<char> stdeq(mydeque);             // stack inicializado con una copia de deque
    cout << "\n\nSTACK DE DEQUE\n\nTamano inicial: " << stdeq.size() << '\n';
    stdeq.push('W');
    stdeq.push('r');
    cout << "\nSe agregan 2 valores, tamano: " << stdeq.size() << "\n\nValores:\n";
    char c;
    int tam = stdeq.size();
    for(int loop=0; loop<tam; loop++)
    {
       c = stdeq.top();
       stdeq.pop();
       cout << "\ttope = " << c << endl;
    };
    cout << endl << endl;

    return 0;
}
fin del ejemplo *****/
```

Algoritmos

Los algoritmos actúan sobre los contenedores. Aunque cada uno provee soporte para sus operaciones básicas, los algoritmos estándar proporcionan el soporte para realizar operaciones más complejas, permitiendo a la vez trabajar sobre distintos tipos de contenedores a la vez. Para su utilización debe incluirse el archivo de cabecera <**algorithm**>.

Todos los algoritmos son templates de funciones, por lo que pueden ser aplicados a cualquier tipo de contenedor. Para un listado completo de los mismos se sugiere consultar la bibliografía específica. Se presentan a continuación los más utilizados.

Cuenta de elementos

Es una de las operaciones básicas. Se puede realizar de dos formas:

template <**class InIter**, **class T**> size_t **count**(**InIter** *start*, **InInter** *end*, **const T** &*val*);

template <**class InIter**, **class UnPred**> size_t **count**(**InIter** *start*, **InInter** *end*, **UnPred** *pfn*);

La primera forma retorna el número de elementos en la secuencia delimitada por *start* y *end* que son iguales a *val*. En la segunda forma el criterio para la cuenta se establece en el predicate unario *pfn* (cuando retorne true).

Eliminación y reemplazo de elementos

Algunas veces es útil generar una nueva secuencia que esté formada por algunos elementos particulares de la secuencia original. Uno de estos algoritmos es **remove_copy**():

template <**class InIter**, **class OutIter**, **class T**>
OutIter remove_copy(**InIter** *start*, **InInter** *end*, **OutIter** *result*, **const T** &*val*);

copia los elementos del rango especificado, eliminando los que son iguales a *val*. El resultado se almacena en la secuencia apuntada por *result* y retorna un iterador a su final. El contenedor de salida debe ser lo suficientemente grande como para almacenar el resultado.

Con **replace_copy**() se puede realizar el reemplazo de los elementos con valor *old* por el valor new en el rango start/end. El resultado se almacena en la secuencia apuntada por result.

template <**class InIter**, **class OutIter**, **class T**> **OutIter replace_copy**
(**InIter** *start*, **InInter** *end*, **OutIter** *result*, **const T** &*old*, **const T** &*new*);

Inversión

El algoritmo **reverse**() permite invertir un rango de valores de una secuencia:

template <**class BiIter**> void **reverse**(**BiIter** *start*, **BiInter** *end*);

```
/***** Ejemplo de count, remove/replace_copy y reverse
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template <class T> void vPrintVec(vector<T> v)
{
   typename vector<T>::iterator ForIter;

   ForIter = v.begin();                    // puntero al primer elemento
   cout << endl;
   while (ForIter != v.end() )             // mientras no sea el Ãºltimo elemento
   {
      cout << *ForIter++ << " ";           // aumento el iterador
   }
   cout << "\n\n";
}

bool dividesBy3(int i)                     // unary predicate: determina si es divisible por 3
{
   bool result;
   (i%3) ? result=false : result=true;

   return result;
}

int main()
{
```

```
    vector<int> iVec;                    // vector de enteros
    vector<char> cVec,                   // vector de caracteres
                 cResult(30);            // vector de 30 elementos
    unsigned loop,
             cnt;

    for(loop=0; loop<30; loop++)
    {
       iVec.push_back(loop);
       cnt = loop;
       if(loop>9)
          cnt/= 10;
       cVec.push_back(cnt + 0x30);
    }
    cout << "Secuencias originales:\n";
    vPrintVec(iVec);
    vPrintVec(cVec);

    cnt = count(iVec.begin(), iVec.end(), 7);
    cout << "\nHay " << cnt << " numeros 7 en la secuencia completa\n\n";
    cnt = count(iVec.begin()+10, iVec.begin()+15, 12);
    cout << "Hay " << cnt << " numeros 12 en la secuencia reducida\n\n";

    cnt = count_if(iVec.begin(), iVec.end(), dividesBy3);
    cout << cnt << " numeros son divisibles por 3.\n\n";

    remove_copy(cVec.begin(), cVec.end(), cResult.begin(), '1');
    cout << "\nSe eliminan los '1' de la secuencia de caracteres\n";
    vPrintVec(cResult);

    replace_copy(cVec.begin(), cVec.end(), cResult.begin(), '2', '3');
    cout << "\nSe reemplazan los '2' por '3' en la secuencia de caracteres\n";
    vPrintVec(cResult);

    reverse(iVec.begin(), iVec.end());
    cout << "\nSecuencia de numeros invertida:\n";
    vPrintVec(iVec);

    string ptr("Prueba de escritura en un string para invertir");
    cout << "\nString original: " << ptr;
    reverse(ptr.begin(), ptr.end());
    cout << "\n\nString invertido: " << ptr << endl << endl;

    return 0;
}
fin del ejemplo *****/
```

Transformación

Mediante **transform**() se pueden modificar los elementos, dentro de un rango, mediante una función provista por el usuario. El resultado se almacena en *result*. Tiene dos formas:

template <**class InIter**, **class OutIter**, **class Func**>
OutIter transform(**InIter** *start*, **InInter** *end*, **OutIter** *result*, **Func** *unaryfunc*);

template <**class InIter1**, **class InIter2**, **class OutIter**, **class Func**> **OutIter transform**
(**InIter1** *start1*, **InInter1** *end1*, **InInter2** *start2*, **OutIter** *result*, **Func** *binaryfunc*);

En la primera forma la función *unaryfunc* recibe el valor de un element como parámetro y retorna el valor transformado. En la segunda forma la función *binaryfunc* recibe como primer parámetro un elemento de la secuencia a ser transformada y como segundo parámetro un elemento de otra secuencia. Ambas funciones retornan un iterador al final de *result*.

```
/***** Ejemplo de transform con funciones unaria y binaria

#include <iostream>
#include <list>
#include <algorithm>
#include <math.h>

using namespace std;

template <class T> void vPrintLst(list<T> lst)
{
   typename list<T>::iterator ForIter;

   ForIter = lst.begin();                    // puntero al primer elemento
   cout << endl;
   while (ForIter != lst.end() )             // mientras no sea el Ãºltimo elemento
   {
      cout << *ForIter++ << " ";             // aumento el iterador
   }
   cout << "\n\n";
}

double reciproco(double i)                   // funcion unaria
{
   double v = 0;
   i ? v = 1.0/i : v;

   return v;                                 // return reciproco
}

double modulo(int i, int j)                  // funcion binaria
{
   return sqrt(i*i + j*j);                   // return suma cuadratica
}

int main()
{
   list<double> vals;
   int loop;

   for(loop=0; loop<10; loop++)
   {
      vals.push_back(double(loop));
   }
   cout << "Contenido original de vals:\n";
   vPrintLst(vals);

   transform(vals.begin(), vals.end(), vals.begin(), reciproco);
   cout << "\nContenido transformado de vals:\n";
   vPrintLst(vals);

   list<int> val1, val2;
   list<double> result(10, 3.5);
```

```
    for(loop=0; loop<10; loop++)
    {
       val1.push_back(loop + 15);
       val2.push_back(loop * 3);
    }
    cout << "\n\nContenidos originales de val1, val2 y result:\n";
    vPrintLst(val1);
    vPrintLst(val2);
    vPrintLst(result);

    transform(val1.begin(), val1.end(), val2.begin(), result.begin(), cuadrado);
    cout << "\nContenido transformado:\n";
    vPrintLst(result);

    return 0;
}
fin del ejemplo *****/
```

Funciones objeto

Son clases que definen **operator()**. Como se mencionó la STL provee funciones básicas <**functional**> y permite al usuario definir las propias. Pueden ser unarias (un solo argumento):

logical_not	negate

o binarias (dos argumentos).

plus	minus	multiplies	divides	modulus
equal_to	not_equal_to	greater	greater_equal	less
less_equal	logical_and	logical_or		

La función objeto más utilizada es less, que determina cuando un objeto es menor que otro. La función negate cambia el signo del elemento. El uso de las funciones objeto en lugar de puntero a función permite la generación de código más eficiente.

Para mantener la flexibilidad cuando se crea una función objeto propia, la STL provee dos clases base que pueden ser heredadas. Las mismas proveen las definiciones de los tipos necesarios, por lo que sólo debe realizarse la sobrecarga de **operator()**.

```
template <class Argument, class Result> struct unary_function
{
   typedef Argument argument_type;
   typedef Result result_type;
};

template <class Argument1, class Argument2, class Result> struct binary_function
{
   typedef Argument1 first_argument_type;
   typedef Argument2 second_argument_type;
   typedef Resu1lt result_type;
};
```

/***** Ejemplo de **transform** con funciones unaria y binaria

```
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
#include <vector>
#include <math.h>
#include "complejos.h"

using namespace std;

template <class T> void vPrintVec(vector<T> v)
{
   typename vector<T>::iterator ForIter;

   ForIter = v.begin();                    // puntero al primer elemento
   cout << endl;
   while (ForIter != v.end() )             // mientras no sea el Ãºltimo elemento
   {
      cout << *ForIter++ << "    ";       // aumento el iterador
   }
   cout << "\n\n";
}

template <class T> void vPrintLst(list<T> v)
{
   typename list<T>::iterator ForIter;

   ForIter = v.begin();                    // puntero al primer elemento
   cout << endl;
   while (ForIter != v.end() )             // mientras no sea el Ãºltimo elemento
   {
      cout << *ForIter++ << "    ";       // aumento el iterador
   }
   cout << "\n\n";
}

class reciproca : unary_function<int, double>
{
   public:
      result_type operator()(argument_type i)
      {
         return (result_type) 1.0/i;       // return reciprocal
      }
};

class modulo : unary_function<complejo, double>
{
   public:
      result_type operator()(argument_type comp)
      {
         return (result_type) sqrt(comp.getReal()*comp.getReal()
                           + comp.getImag()*comp.getImag());
      }
};

class promedio : binary_function<int, char, double>
{
   public:
      result_type operator()(first_argument_type factor, second_argument_type c)
```

```
        {
          return (result_type) ( (factor + double(c))/2.0 );
        }
};

int main()
{
   list<int> iVals;
   list<double> dVals(10);
   int loop;
   for(loop=1; loop<11; loop++)
   {
      iVals.push_back(loop);
   }
   cout << "Contenido original de iVals:\n";
   vPrintLst(iVals);

   transform(iVals.begin(), iVals.end(), dVals.begin(), reciproca());
   cout << "Valores reciprocos con mi funcion unaria reciproca:\n";
   vPrintLst(dVals);
   transform(iVals.begin(), iVals.end(), dVals.begin(), negate<int>());
   cout << "Valores negados con stl negate<int>():\n";
   vPrintLst(dVals);

   complejo c1(1.1, 2.2),          // se crean e inicializan dos complejos (constructor general)
            c2(3.3, 4.4),          // se crea un complejo con el constructor por defecto
            c3(7.5),               // se crea un complejo con el valor por defecto (0.0) del 2do. argumento
            suma = c1 + c2,        // se crea un complejo a partir del resultado (constructor de copia)
            producto = c1 * c3;    // inicializados por el constructor por defecto

   vector<complejo> cVec(3);       // crea un vector de 3 complejos
   vector<double> dVec(5);         // crea un vector de 5 doubles
   cVec[0] = c1;                   // asigna valores consecutivos
   cVec[1] = c2;
   cVec[2] = c3;
   cVec.push_back(suma);           // agrego valores al final, crece solo
   cVec.push_back(producto);
   cout << "\n\n\nVector de " << cVec.size() << " complejos:" << endl;
   vPrintVec(cVec);
   transform(cVec.begin(), cVec.end(), dVec.begin(), modulo());
   cout << "Calculo de los modulos con mi funcion unaria:\n";
   vPrintVec(dVec);

   vector<int> iFact;
   vector<char> cFact;
   for(loop=0; loop<5; loop++)
   {
      iFact.push_back(27-2*loop);
      cFact.push_back(loop + 'A');
   }
   transform(iFact.begin(), iFact.end(), cFact.begin(), dVec.begin(), promedio());
   cout << "\n\n\nCalculo del promedio entre int y char con mi funcion binaria:\n";
   vPrintVec(iFact);
   vPrintVec(cFact);
   vPrintVec(dVec);
   cout << endl << endl;

  return 0;
}
fin del ejemplo *****/
```