UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNLVIRTUAL

Programación para videojuegos II

Unidad 1
Templates y STL

Docente
Pablo Abratte

CONTENIDOS

Introducción

En esta unidad aprenderemos a utilizar una herramienta indispensable para cualquier programador del lenguaje C++: la *Standard Template Library* o STL. La misma nos provee de un conjunto de estructuras y algoritmos genéricos que pueden utilizarse con cualquier tipo de dato predefinido o creado por el usuario.

Antes de adentrarnos en el estudio de esta útil biblioteca, es conveniente comprender los mecanismos que posibilitan su funcionamiento: los *templates* o plantillas.

1. Programación genérica y templates en C++

La programación genérica es un estilo de programación centrada en la lógica de los algoritmos más que en los tipos de datos que intervienen en los mismos. El objetivo es maximizar la generalidad y reutilización de los procedimientos, haciendo que los algoritmos acepten tipos de datos como parámetro.

Programación genérica
Es un estilo de programación centrada en la lógica de los algoritmos más que en los tipos de datos que intervienen en los mismos. El objetivo es maximizar la generalidad y reutilización de los procedimientos, haciendo que los algoritmos acepten tipos de datos como parámetro.

En el lenguaje C++, la programación genérica es posible gracias a los templates o plantillas, que nos permiten desarrollar funciones (o también clases) para ser utilizadas con cualquier tipo de dato, ya sea un tipo predefinido del lenguaje (`int`, `float`) o un tipo definido por el usuario (struct o clase).

Para ilustrar mejor la necesidad de utilizar templates, veremos a continuación un ejemplo sencillo:

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void BubbleSort(int arr[], int n){
5.      int i,j;
6.      int min;
7.      for(i=0; i<n-1; i++){ // bucle principal
8.          for(j=0; j<n-1; j++){
9.              if(arr[j]>arr[j+1]){
10.                 min=arr[j];
11.                 arr[j]=arr[j+1];
12.                 arr[j+1]=min;
13.             }
14.         }
15.     }
16. }
17.
18. int main(int argc, char *argv[]){
19.     int vector1[50];
20.     float vector2[50];
21.     double vector3[50];
22.
23.     BubbleSort(vector1, 50);
24.     BubbleSort(vector2, 50);
25.     BubbleSort(vector3, 50);
26.
27.     return 0;
28. }
```

En *Programación de Videojuegos I* conocimos el algoritmo de ordenamiento por burbuja que se muestra en el código anterior. El programa falla al compilar, ya que el algoritmo sólo está definido para arreglos del tipo `int`, por lo que no resulta posible utilizarlo con `vector1` y `vector2`, que son de tipo `float` y `double`, respectivamente. Una solución es escribir dos sobrecargas de la función `BubbleSort` que trabajen con estos tipos, lo cual implicaría tener tres funciones que realicen el mismo procedimiento y sólo difieran en el tipo de arreglo aceptado.

Una mejor forma de solucionar este inconveniente es recurrir al uso de templates para expresar el algoritmo en función de un tipo de dato. El resultado puede observarse a continuación:

Templates
Son plantillas que se utilizan para expresar un algoritmo en función de un tipo de dato.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  template <class T>
5.  void BubbleSort(T arr[], int n){
6.      int i,j;
7.      T min;
8.      for(i=0; i<n-1; i++){ // bucle principal
9.          for(j=0; j<n-1; j++){
10.             if(arr[j]>arr[j+1]){
11.                 min=arr[j];
12.                 arr[j]=arr[j+1];
13.                 arr[j+1]=min;
14.             }
15.         }
16.     }
17. }
18.
19. int main(int argc, char *argv[]){
20.     int vector1[50];
21.     float vector2[50];
22.     double vector3[50];
23.
24.     BubbleSort(vector1, 50);
25.     BubbleSort(vector2, 50);
26.     BubbleSort(vector3, 50);
27.
28.     return 0;
29. }
```

Si intentamos nuevamente compilar el programa de ejemplo, veremos que esta vez no ocurren errores. La función `BubbleSort` ha sido templatizada y ahora puede ser usada tanto para los tipos `int`, como `float` y `double`.

Para lograr esto, hemos parametrizado al algoritmo con un tipo de dato genérico al cual nos referiremos como *T*. En la línea 4, advertimos al compilador que la función que implementaremos a continuación será una plantilla y estará expresada en función de un tipo de dato *T* que no conocemos. Dentro de la función, también utilizaremos al tipo *T* para denominar a las variables, cuyo tipo deba ser también genérico, en este caso, la variable *min.*

El compilador no puede convertir una plantilla en código máquina, ya que al desconocer el tipo genérico, es imposible saber de antemano la cantidad de memoria que se necesita reservar. Sin embargo, al analizar el resto del código, se dará cuenta de que nuestra función templatizada es llamada, primero, con un arreglo de ints; luego, con uno de floats, y después, con uno de doubles.

Posteriormente, a partir de la plantilla, creará tres copias de la función, reemplazando en cada copia a *T* por cada uno de los tipos de datos, y compilará luego las copias de la plantilla con sus tipos definidos. De esta manera, en realidad, tendremos tres copias de la misma función, pero habremos escrito el código sólo una vez y el compilador hará el resto del trabajo por nosotros, aplicando el template para los tipos con los que utilicemos la función.

Como observaciones extras respecto al ejemplo, podemos notar que no es necesario que todos los parámetros pasados a la función sean de tipo genérico (puede ocurrir que ningún parámetro sea genérico, pero sí alguna variable interna o el tipo de retorno). Además, aunque *T* es el identificador utilizado con más frecuencia para referirse a un tipo genérico, puede ser reemplazado por cualquier identificador válido.

No solamente las funciones pueden ser templatizadas, sino también las clases. Ilustraremos esto utilizando nuevamente un ejemplo.

Deseamos modelar, mediante una clase llamada Rect, un rectángulo que represente una región de la pantalla o de nuestro nivel. Como atributos proponemos las coordenadas *x* e *y* del vértice superior izquierdo del rectángulo, su ancho y su alto, según pueden observarse en la *Figura 1.*

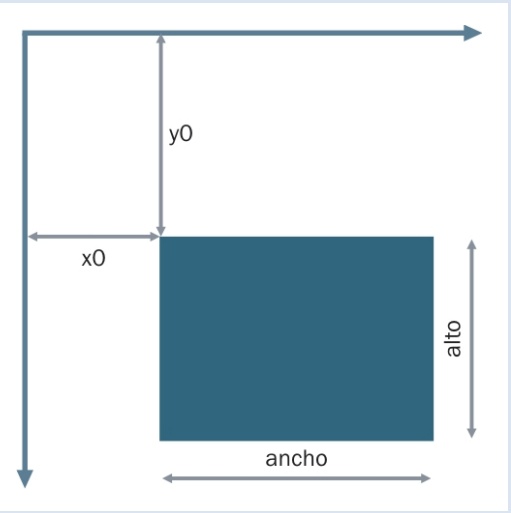

Fig 1. Atributos del objeto rectángulo.

El código correspondiente se muestra a continuación:

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Rect{
5.  private:
6.      int x0, y0, ancho, alto;
7.
8.  public:
9.      Rect(int x0, int y0, int ancho, int alto);
10.     int GetAncho();
11.     int GetAlto(){return alto;};
12. }
13.
14. Rect::Rect(int x0, int y0, int ancho, int alto){
15.     this->x0=x0;
16.     this->y0=y0;
17.     this->ancho=ancho;
18.     this->alto=alto;
19. }
20.
21. int Rect::GetAncho(){return ancho;}
22.
23. int main(int argc, char *argv[]){
24.     Rect r(1, 2, 3, 4);
25.     cout<<r.GetAlto()<<" "<<r.GetAncho()<<endl;
26.     return 0;
27. }
```

Como mencionamos anteriormente, la clase Rect puede resultarnos útil para representar regiones de nuestra pantalla compuestas de cantidades discretas, como los píxeles, pero si tuviésemos la necesidad de utilizar esta clase para referirnos a regiones del mundo virtual en el que transcurren los niveles de nuestros juegos, seguramente perderíamos precisión al utilizar regiones representadas por cantidades enteras y no reales. Nuevamente nos encontramos en un predicamento similar al anterior, el cual podemos superar utilizando templates para definir la clase de manera genérica, como indica el siguiente código:

```cpp
#include <iostream>
using namespace std;

template <typename T>
class Rect{
private:
    T x0, y0, ancho, alto;

public:
    Rect(T x0, T y0, T ancho, T alto);
    T GetAncho();
    T GetAlto(){return alto;};
};

template <typename T>
Rect<T>::Rect(T x0, T y0, T ancho, T alto){
    this->x0=x0;
    this->y0=y0;
    this->ancho=ancho;
    this->alto=alto;
}

template <class T>
T Rect<T>::GetAncho(){return ancho;}


int main(int argc, char *argv[]) {
    Rect<float> r1(0.1, 0.2, 0.3, 0.4);
    Rect<int> r2(1, 2, 3, 4);

    cout<<r1.GetAlto()<<" "<<r2.GetAncho()<<endl;

    return 0;
}

```

En el ejemplo se observa que para crear una plantilla de clase se antepone la declaración `template <typename T>` a la declaración de la clase- Luego, pueden declararse atributos y funciones que utilicen valores genéricos. También es posible ver que, a diferencia del ejemplo anterior, en la declaración esta vez se utiliza la palabra *typename* en lugar de *class.* Ambas formas son equivalentes y queda a criterio del programador utilizar la que le resulte más cómoda para templatizar, tanto funciones como clases.

Para las funciones miembros de Rect que están definidas fuera del ámbito de la clase, como su constructor y `GetAncho()`, es necesario incluir nuevamente la declaración de template y agregar, además, `<T>` al nombre de la clase. Esto último es necesario debido a que el compilador creará diversas copias de la clase y sus miembros para distintos tipos de datos, y las funciones deberán identificar de cuál de esas copias son miembros.

Finalmente, al observar la forma en que los objetos son instanciados, aparecen nuevas diferencias. En el ejemplo anterior, el compilador sabía a cuál de las copias de la plantilla debía llamar, observando el tipo de los parámetros pasados a la función. A la hora de crear una instancia de una clase templatizada, esto no es posible, por lo que resulta necesario explicitarle al compilador el tipo de dato que utilizaremos. El compilador usará esta información para saber cuáles son las copias de la plantilla que debe generar. Luego de especificado el tipo, a la hora de construir el objeto, las llamadas a los métodos pueden realizarse de forma normal.

El ejemplo realizado es similar a la clase templatizada `sf::Rect<T>` presente en la librería SFML.

Dijimos anteriormente que una plantilla podía recibir tipos de datos definidos por el usuario. En este punto intentaremos llegar un paso más lejos e imaginar la posibilidad de pasar un arreglo de `Rect<float>` a la función templatizada `BubbleSort`. Recordemos que el mecanismo de templatización consiste en el compilador creando

una copia de la plantilla, reemplazando *T* por `Rect<float>`. Sin embargo, la correcta sintáxis y semántica de estas construcciones no siempre está garantizada. En el código resultante se comparan dos objetos de tipo `Rect<float>` con el operador < (línea 10 en el código de la función `BubbleSort` templatizada). Como esta operación no ha sido definida, la compilación fallará.Esto nos lleva a limitar el uso de una plantilla a cualquier tipo de dato que soporte las manipulaciones que el algoritmo o clase hace con las variables. En el caso de la función templatizada `BubbleSort`, estas operaciones son la asignación y la comparación mediante el operador < entre dos variables de tipo `Rect`.

Como una desventaja de las plantillas podemos mencionar la dificultad de rastrear los errores de compilación, debido a la poca legibilidad de las salidas producidas por el compilador en situaciones como la mencionada en el párrafo anterior. Otra desventaja aún peor es, como se mencionó anteriormente, la imposibilidad de traducir una plantilla a código máquina, haciendo necesario volver a compilar todo el código templatizado cada vez que se modifica alguna parte del programa que la utiliza. La sobrecarga producida puede aumentar el tiempo de compilación de manera considerable, en el caso de bibliotecas templatizadas de gran tamaño.

La templatización no está necesariamente limitada a la utilización de un solo tipo genérico, sino que pueden crearse plantillas que dependan de dos o más tipos genéricos. Tendremos oportunidad de observar esto cuando estudiemos algunas estructuras de la biblioteca STL.

Para ahondar en los conceptos tratados y conocer las características avanzadas de las plantillas en C++, pueden consultar *The C++ Programming Language,* de Bjarne Stroustrup.

2. Standard Template Library

La biblioteca STL o Standard Template Library implementa estructuras de datos y algoritmos de forma templatizada para que puedan ser usados con cualquier tipo de datos predefinidos en C++, o incluso con tipos definidos por el usuario.

Standard Template Library
También llamada biblioteca STL, implementa estructuras de datos y algoritmos de forma templatizada para que puedan ser usados con cualquier tipo de datos predefinidos en C++, o incluso con tipos definidos por el usuario.

Dicha biblioteca fue desarrollada, en gran parte, por Alexander Stepanov. Stepanov fue un precursor de la corriente que hoy se conoce como programación genérica, cuyas ideas empezó a desarrollar en 1979. En 1987 comenzó a buscar una forma de portar características de programación genérica, que ya estaban presentes en el lenguaje Ada a C++, aprovechando el modelo computacional de este último y su manejo flexible de memoria para ganar generalidad y eficiencia. La investigación de Stepanov y sus colaboradores fue llevada a cabo en los laboratorios de AT&T, primeramente, y en los de Hewlett-Packard, más tarde. Finalmente, en 1994, STL fue incorporada a la versión ANSI/ISO de C++.

Realizar una descripción completa de la librería es una tarea complicada e innecesaria. Por lo tanto, centraremos nuestro análisis en tres elementos claves de la biblioteca: los *contentenedores*, los *iteradores* y los *algoritmos*.

Los *contenedores,* como su nombre lo indica, se encargan del almacenamiento y la organización en memoria de los datos. Consisten en clases templatizadas que implementan estructuras como vectores, listas, pilas, colas, etc.

Los *iteradores* son objetos cuyo comportamiento es similar al de un puntero. Nos permiten movernos dentro de los contenedores y hacer referencia a una posición de memoria.

Los *algoritmos* son funciones genéricas que nos permiten procesar los datos guardados en los contenedores. Proveen algunas funcionalidades como inicializar, buscar, ordenar y copiar datos de los contenedores.

Nuevamente abordaremos la temática desde un ejemplo sencillo, que muestra la utilización básica de un vector y una lista STL:

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  #include <vector>
4.  #include <list>
5.
6.  using namespace std;
7.
8.  int main(int argc, char *argv[]) {
9.      // declaramos la lista y el vector
10.     list<float> l;
11.     vector<int> v(10, 26);
12.
13.     // inicializamos el generador de numeros aleatorios
14.     srand(time(NULL));
15.
16.     // cargamos 10 valores aleatorios en la lista (entre 0 y 1)
17.     for(unsigned i=0; i<10; i++){
18.         l.insert(l.end(), rand()/float(RAND_MAX));
19.     }
20.
21.     // mostramos los valores del vector
22.     cout<<"Valores del vector: "<<endl;
23.     for(unsigned i=0; i<v.size(); i++){
24.         cout<<v[i]<<endl;
25.     }
26.
27.     // mostramos los valores de la lista
28.     cout<<"Valores de la lista: "<<endl;
29.     list<float>::iterator p=l.begin();
30.     while(p!=l.end()){
31.         cout<<*p<<endl;
32.         p++;
33.     }
34.
35.     return 0;
36. }
37.
```

En las líneas 3 y 4 se incluyen las respectivas cabeceras para trabajar con estas estructuras. La creación de los objetos *lista* y *vector* es realizada en las líneas 10 y 11.Como se trata de clases templatizadas, es necesario especificar el tipo de dato deseado.

En el caso de la lista, observamos que el constructor no lleva parámetros, en tanto que en el caso del vector, los dos parámetros especifican el tamaño inicial del mismo (ya que se trata de un arreglo dinámico) y el valor con el cual se inicializará cada una de sus posiciones en memoria. En este caso, tendremos un arreglo de 10 enteros inicializados con el valor 26.

El bucle `for` de la línea 17 inserta 10 valores aleatorios en la lista, utilizando la función miembro `insert()` que recibe la posición y el valor a insertar. Para indicar que cada nuevo valor debe insertarse al final de la lista, se utiliza la función miembro `end()`, la cual devuelve un iterador (o puntero) al final de la lista.

Los bucles de las líneas 23 y 28 se encargan de mostrar los elementos de los dos contenedores. En el caso del vector, vemos que sus elementos pueden ser accedidos mediante el operador [], como si se tratara de un arreglo común. Además, observamos la utilización de la función `size()` para limitar la cantidad de iteraciones del bucle. Dicha función devuelve la cantidad de elementos en la estructura y está también disponible para la mayoría de los demás contenedores.

La lista es recorrida mediante un iterador declarado en la línea 29. Como vemos, el tipo de iterador utilizado está definido dentro de la clase `list<float>`. Para inicializarlo, se utiliza la función miembro `begin()`, que devuelve un iterador al primer elemento de la lista. A partir de allí, puede recorrerse la lista incrementando el iterador y aplicándole el operador de desreferencia para obtener el dato apuntado, como si se tratara de un

puntero común y corriente. El bucle continuará ejecutándose mientras nuestro iterador no alcance la posición final devuelta por `end()`.

Cabe destacar que dicha función devuelve un iterador a la posición posterior al último elemento contenido. En esta posición es posible insertar un dato, ya que representa la primera posición libre después del último elemento de la estructura. Sin embargo, no es correcto desreferenciar este iterador, dado que dicha posición no contiene un dato real. De la misma forma, hubiese sido posible recorrer al vector mediante iteradores.

A continuación desarrollaremos, de manera más formal, los componentes de la biblioteca.

2.1 Contenedores

Contenedores
Son clases templatizadas que tienen como función proveer una estructura para el almacenamiento de los datos y funciones para fin de manipularlos. STL provee aproximadamente 15 contenedores diferentes.

Como se mencionó anteriormente, los contenedores son clases templatizadas que tienen como función proveer una estructura para el almacenamiento de los datos, y funciones para manipularlos. STL provee aproximadamente 15 contenedores diferentes. En esta sección presentaremos solamente los que nos resultarán de mayor utilidad.

Vector

Representa un arreglo similar a los usados hasta el momento, pero con algunas mejoras. Al igual que los arreglos estáticos, sus elementos son guardados en posiciones contiguas de memoria, lo cual permite acceder a ellos de manera aleatoria, no sólo a través de iteradores, sino también a partir de un desplazamiento (u offset) respecto de la posición inicial.

Sin embargo, a diferencia de los arreglos estáticos, el vector STL puede variar automáticamente su tamaño en caso de ser necesario. Dado que sería ineficiente reservar un nuevo bloque contiguo de memoria y mudar todos los datos al mismo cada vez que se realiza una nueva inserción o remoción, cuando la memoria se agota, el vector reserva un nuevo bloque con lugar extra para varias inserciones futuras. De esta forma, las reservas de memoria son menos frecuentes y se gana eficiencia. Internamente, cada vector mantiene dos variables con la cantidad real de elementos (*size*) y la cantidad total de memoria disponible (*capacity*). El objeto provee funciones para que los ajustes de tamaño puedan ser realizados por el usuario.

Como observamos en el ejemplo anterior, para hacer uso de este contenedor, es necesario incluir la cabecera <vector>. A continuación, compendiaremos algunas de las funciones más útiles de esta clase.

```
vector<T>();
```

Constructor por defecto; crea un vector sin memoria reservada para elementos.

```
vector<T>(size_type n, const T& value);
```

Constructor: crea un vector de tamaño *n* e inicializa todos sus elementos con el valor *value*.

```
vector<T>& operator=(const vector<T>& x);
```

Permite asignar o crear una copia de objetos de tipo vector.

```
vector<T>::iterator begin();
```

Retorna un iterador al primer elemento del vector.

```
vector<T>::iterator end();
```

Devuelve un iterador a la posición final del vector (posición posterior al último valor).

```
size_type size();
```

Retorna la cantidad de elementos del vector.

```
bool empty();
```

Devuelve *verdadero* si el vector está vacío, o *falso,* si ocurre lo contrario.

```
void resize(size_type sz);
```

Permite redimensionar manualmente la memoria del arreglo al nuevo tamaño *sz*. Si *sz* es menor que el tamaño actual, los elementos sobrantes se perderán.

```
T &operator[](size_type n);
```

Devuelve una referencia al elemento del vector en la posición *n*. El método no controla que el valor *n* se encuentre dentro de los límites de la memoria, sino que es responsabilidad del usuario hacerlo. Sirve tanto para leer como para escribir los elementos, por lo que nos permite utilizar el vector como un arreglo estático común.

```
vector<T>::iterator insert(vector<T>::iterator position,
                                        const T& x);
```

Insertar un elemento *x* en la posición *position,* agrandando el tamaño del vector, sí es necesario. Devuelve un iterador a la posición del elemento insertado.

```
vector<T>::iterator erase(vector<T>::iterator position);
vector<T>::iterator erase(      vector<T>::iterator first,
                                vector<T>::iterator last);
```

Permite eliminar del vector el elemento en la posición *position* o los elementos en el rango [*first*, *last*). Devuelve un iterador a la posición siguiente del último de los elementos eliminados.

```
void clear();
```

Elimina todos los elementos del vector, dejándolo vacío.

```
void push_back(const T& x);
```

Inserta un elemento con valor *x* al final del vector, redimensionándolo si es necesario. Es equivalente a v.insert(v.end(), x).

```
void pop_back(const T& x);
```

Quita el último elemento del vector. Es equivalente a `v.erase(v.end())`.

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  #include <vector>
4.
5.  using namespace std;
6.
7.  int main(int argc, char *argv[]){
8.      vector<float> v;
9.
10.     // inicializamos el generador de numeros aleatorios
11.     srand(time(NULL));
12.
13.     // insertamos 10 valores aleatorios en el vector
14.     for(unsigned i=0; i<10; i++){
15.         v.push_back(rand()/float(RAND_MAX));
16.     }
17.
18.     // mostramos los valores del vecor
19.     for(vector<float>::iterator p=v.begin(); p<v.end(); p++){
20.         cout<<*p<<endl;
21.     }
22.
23.     return 0;
24. }
```

A fin de afianzar los contenidos que hemos visto hasta ahora, estudiaremos un ejemplo similar al anterior, mostrando algunas formas alternativas de operar sobre un vector STL.

Esta vez, el vector creado contiene elementos de tipo `float`. Al utilizar el constructor por defecto, no se reserva memoria alguna para guardar datos, por lo que para insertarlos debe usarse una función como `insert()` o `push_back()`, los cuales realizan de forma automática la reserva de memoria según se necesite. Además, se utiliza una forma alternativa de recorrer el vector con iteradores.

List

Se trata de una lista doblemente enlazada donde los elementos no son almacenados de forma contigua, como en un vector, sino que cada uno tiene su propio bloque de memoria independiente y punteros al elemento anterior y al siguiente. Esto permite reservar memoria individual para cada elemento en cada inserción y remoción. La estructura es de acceso secuencial, por lo que para acceder, por ejemplo, al sexto elemento, es necesario recorrer primero los cinco elementos anteriores. Comparadas con los vectores, las listas son más eficientes cuando se trata de inserciones y remociones, mientras que los vectores son mejores para recorrer sus elementos, ya que los mismos están contiguos en memoria y permiten un acceso aleatorio.

Para utilizar la lista STL es necesario incluir la cabecera <list>. Al igual que lo hicimos con el vector, expondremos una breve referencia de sus funciones.

```
list<T>();
```

Constructor por defecto, crea una lista vacía.

```
list<T>(size_type n, const T& value);
```

Constructor: crea una lista con *n* elementos inicializados con el valor *value*.

```
list<T>& operator=(const list<T>& x);
```

Permite asignar o crear una copia de objetos de tipo list.

```
list<T>::iterator begin();
```

Retorna un iterador al primer elemento de la lista.

```
list<T>::iterator end();
```

Devuelve un iterador a la posición final de la lista (posición posterior al último valor).

```
size_type size();
```

Retorna la cantidad de elementos de la lista.

```
bool empty();
```

Devuelve *verdadero* si la lista está vacía, o *falso,* cuando ocurre lo contrario.

```
list<T>::iterator insert(       list<T>::iterator position,
                                const T& x);
```

Insertar un elemento *x* en la posición *position*. Devuelve un iterador a la posición del elemento insertado.

```
list<T>::iterator erase(list<T>::iterator position);
list<T>::iterator erase(        list<T>::iterator first,
                                list<T>::iterator last);
```

Permite eliminar del vector el elemento en la posición *position* o los elementos en el rango [*first*, *last*). Devuelve un iterador a la posición siguiente del último de los elementos eliminados.

```
void clear();
```

Elimina todos los elementos de la lista, dejándola vacía.

```
void push_back(const T& x);
```

Inserta un elemento con valor *x* al final de la lista. Es equivalente a `l.insert(l.end(), x)`.

```
void pop_back(const T& x);
```

Quita el último elemento de la lista. Es equivalente a `l.erase(l.end())`.

```
void push_front(const T& x);
```

Inserta un elemento con valor *x* al principio de la lista. Es equivalente a `l.insert(l.begin(), x)`.

```
void pop_front(const T& x);
```

Quita el primer elemento de la lista. Es equivalente a `l.erase(l.begin())`.

Más adelante veremos que la librería STL provee algoritmos genéricos que pueden ser utilizados con diversos contenedores. Sin embargo, muchos de ellos necesitan que los contenedores permitan determinado tipo de acceso a los datos. En el caso de la lista, que no provee acceso aleatorio sino secuencial, muchos de esos algoritmos genéricos no pueden aprovecharse. Por eso, para suplir esta necesidad, el contenedor debe implementarlos en sus propias rutinas. Veamos algunas a continuación.

```
void splice(list<T>::iterator position, list<T>& x);
void splice(    list<T>::iterator position,
                list<T>& x,
                list<T>::iterator first,
                list<T>::iterator last);
```

La primera versión de esta función mueve los elementos de la lista *x* a la posición *position* de la lista actual. La segunda versión permite realizar la misma operación, pero no sobre la lista completa, sino en el rango *[first, last).*

```
void unique();
```

Elimina de la lista los elementos repetidos, dejando sólo la primera ocurrencia de cada valor. Por ejemplo, dada una lista con los elementos (21, 5, 10, 20, 20, 5), luego de aplicar `unique()`, se obtiene (21, 5, 10, 20).

```
void sort();
```

Ordena la lista.

```
void merge(list<T> &x);
```

Mezcla la lista actual con la lista *x,* manteniendo el orden. Ambas listas deben estar previamente ordenadas. Por ejemplo, dadas las listas A=(1, 2, 3, 4) y B=(3, 4, 5, 6), luego de hacer `A.merge(B)`, A=(1, 2, 3, 3, 4, 4, 5, 6).

```
void reverse();
```

Invierte el orden de los elementos en la lista.

Nuevamente presentaremos un ejemplo, a fin de poder analizar algunas cuestiones prácticas:

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  #include <list>
4.
5.  using namespace std;
6.
7.  int main(int argc, char *argv[]){
8.      list<int> l;
9.
10.     // inicializamos el generador de numeros aleatorios
11.     srand(time(NULL));
12.
```

```
13.      // insertamos 20 valores aleatorios en la lista
14.      for(unsigned i=0; i<20; i++){
15.          l.insert(l.end(), rand()%10);
16.      }
17.
18.      cout<<"La lista:";
19.      list<int>::iterator p=l.begin();
20.      while(p!=l.end()){
21.          cout<<" "<<*p;
22.          p++;
23.      }
24.      cout<<endl;
25.
26.      l.sort();
27.      cout<<"La lista ordenada:";
28.      for(p=l.begin(); p!=l.end(); p++){
29.          cout<<" "<<*p;
30.      }
31.      cout<<endl;
32.
33.      l.unique();
34.      cout<<"La lista sin repetidos:";
35.      for(p=l.begin(); p!=l.end(); p++){
36.          cout<<" "<<*p;
37.      }
38.      cout<<endl;
39.
40.      return 0;
41.  }
```

El ejemplo ilustra la utilización de algunas funciones de la clase list y formas alternativas de recorrerla usando iteradores. Notemos que, al comparar iteradores de listas, se usa el operador != y no el operador <, como en el caso de vectores. Al ser el iterador un puntero y al estar los valores de la lista en posiciones de memoria independientes, no es correcto utilizar el operador < para comparar dos iteradores de listas, ya que un elemento cuya posición sea posterior a otro en la lista no necesariamente tendrá una posición de memoria posterior.

Stack

Se trata de una pila o estructura de tipo LIFO (el último en entrar será el primero en salir).

Internamente, la clase stack se implementa heredando de la clase más general, como list o vector, y creando una interface que permita al usuario acceder solamente a algunas de sus funciones. Por ello stack es considerada un adaptador a otro contenedor más general (Adaptor Container). Para utilizarla, es necesario incluir la cabecera <stack>.

Sus funciones son las siguientes:

```
stack<T>();
```

Constructor por defecto, crea una pila vacía.

```
bool empty();
```

Devuelve *verdadero* si la pila está vacía, o falso, si ocurre lo contrario.

```
T top();
```

Devuelve el elemento en el tope de la pila.

```
void push(const T &x);
```

Inserta el elemento *x* en el tope de la pila.

```
void pop();
```

Quita el elemento que está en el tope de la pila.

Queue

Es una estructura de tipo FIFO (el primero en entrar será el primero en salir) o cola. Al igual que la clase stack, se trata de un adaptador a otro contenedor de mayor funcionalidad. Para utilizarla, es necesario incluir la cabecera <queue>. STL también provee la clase deque, que implementa una doble cola, aunque no la trataremos en esta materia.

A continuación, enumeramos las funciones de la clase queue:

```
queue<T>();
```

Constructor por defecto, crea una cola vacía.

```
bool empty();
```

Devuelve *verdadero* si la cola está vacía, o *falso,* si ocurre lo contrario.

```
T front();
```

Devuelve el elemento del frente de la cola, es decir, el próximo por salir.

```
T back();
```

Devuelve el elemento del final de la cola, esto es, el último en entrar.

```
void push(const T &x);
```

Inserta el elemento *x* final de la cola.

```
void pop();
```

Quita el elemento que está en el frente de la cola.

Map

El *mapeo* (map) es un contenedor asociativo donde cada elemento es una combinación de dos valores: un valor clave o key y un valor mapeado a dicha clave. La clave es utilizada para identificar unívocamente a cada elemento, por lo que en un mapeo no puede haber dos elementos que tengan la misma clave. El contenedor multipap, que no estudiaremos aquí, provee una funcionalidad similar, permitiendo claves repetidas.

La mayoría de sus funciones son similares a las de un vector. Sin embargo, está optimizado para realizar búsquedas de manera eficiente.

La clase es una plantilla con dos tipos parametrizados y cada elemento es guardado dentro de otro contenedor más simple: `pair<class T, class S>`. El mismo es un contenedor genérico que tiene un par de datos de tipo T y S. Los atributos `first` y `second` permiten acceder al primer y segundo componente del par.

El manejo de esta clase merece una explicación profunda, ya que se trata de una estructura más compleja que las anteriormente vistas. Por esta razón, luego de presentar sus métodos, veremos un ejemplo de cómo se utiliza este contenedor.

```
map<class T, class S> map();
```

Constructor por defecto; crea un mapeo vacío con claves de tipo T y valores de tipo S.

```
map<class T, class S>& operator=(const map<class T, class S>& x);
```

Permite asignar o realizar una copia entre mapeos.

```
map<class T, class S>::iterator begin();
```

Devuelve un iterador al comienzo del mapeo.

```
map<class T, class S>::iterator end();
```

Devuelve un iterador a la posición final del mapeo.

```
size_type size();
```

Devuelve la cantidad de elementos contenidos en el mapeo.

```
bool empty();
```

Devuelve *verdadero* si el mapeo está vacío, o *falso,* si ocurre lo contrario.

```
pair<map<class T, class S>::iterator, bool>
insert(const pair<class T, class S>& x );
```

Inserta un nuevo elemento en el mapeo. El mismo consiste en un par formado por una clave de tipo T y un valor de tipo S, modelado mediante otro tipo de contenedor: `pair<class T, class S>`.

Como dijimos anteriormente, el mapeo no permite la existencia de claves duplicadas. Por lo tanto, el elemento no se insertará en el mapeo en caso de que ya exista otro con la misma clave y tampoco se modificará el valor mapeado existente para la clave.

La función devuelve otro par de valores, compuesto por un iterador a la posición del elemento insertado y un booleano, el cual será *verdadero* si el valor ha sido insertado en el mapeo, o *falso,* en caso contrario.

```
map<class T, class S>::iterator
erase(map<T, S>::iterator position);
map<class T, class S>::iterator
erase(map<T, S>::iterator first, map<T, S>::iterator last);
```

Tal como sucede en las estructuras anteriores, existen sobrecargas para eliminar un solo elemento o un rango de ellos.

```
void clear ( );
```

Elimina todos los elementos del mapeo dejándolo vacío.

```
map<class T, class S>::iterator find(T &key);
```

Busca el elemento, cuya clave es igual a `key`, y devuelve un iterador al mismo, en caso de haberlo encontrado, o `map::end()`, si ocurre lo contrario.

```
S &operator[](T &key);
```

Devuelve el valor mapeado asociado a la clave `key`. Si no existe un valor con dicha clave, el mapeo inserta un nuevo elemento con clave `key` y un valor mapeado inicializado por defecto.

El siguiente ejemplo permite manejar los nombres y las notas de un grupo de alumnos mediante un mapa:

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

using namespace std;

int main(int arcg, char *argv[]){
    map<string, int> calificaciones;
    string nombre;
    int nota;

    // leemos por consola los nombres y calificaciones
    // de 5 alumnos y los insertamos en el mapa
    for(unsigned i=0; i<5; i++){
        cout<<"Ingrese el nombre del alumno: ";
        cin>>nombre;
        cout<<"Ingrese la calificacion: ";
        cin>>nota;
        // insertamos en el mapeo un nuevo par
        // con el nombre y la nota del alumno
        calificaciones.insert(  calificaciones.end(),
                                pair<string, int>(nombre, nota));
    }
    cout<<endl;

    // busca si existe la calificacion de un alumno
    map<string, int>::iterator f;
```

```
27.      f=calificaciones.find("Kenny Mathews");
28.      if(f!=calificaciones.end()){
29.          cout<<"La calificacion de Kenny Mathews es: ";
30.          cout<<f->second<<endl;
31.      }else{
32.          cout<<"NO EXISTE calificacion para Kenny Mathews"<<endl;
33.      }
34.      cout<<endl;
35.
36.      // con esto podemos insertar una nueva calificacion
37.      // o cambiar una que ya existe
38.      calificaciones["Shannon Mathews"]=5;
39.
40.
41.      // finalmente mostramos todos los elementos del mapeo
42.      cout<<"CALIFICACIONES"<<endl;
43.      map<string, int>::iterator p=calificaciones.begin();
44.      pair<string, int> miPar;
45.      while(p!=calificaciones.end()){
46.          miPar=*p;
47.          cout<<"Alumno: "<<miPar.first;
48.          cout<<"\t\t\t\tNota: "<<miPar.second<<endl;
49.          p++;
50.      }
51.
52.
53.      return 0;
54.  }
```

Para almacenar los datos de los alumnos, en la línea 7 se crea un mapa a fin de asociar el nombre de tipo `string` (cadena de texto) con la nota (`int`).

A continuación, el usuario debe ingresar el nombre de cada alumno con su respectiva nota. Como mencionamos anteriormente, cada elemento del mapa es otro contenedor de tipo `pair`, por lo que necesitamos componer un contenedor del mismo con los dos datos de cada alumno para insertarlo después en el mapa (líneas 20 y 21).

Luego, se pueden realizar búsquedas mediante el método `find()` y, en caso de obtener un iterador válido, acceder a los miembros del par mediante los atributos `first` y `second`.

En la línea 38 es posible observar cómo puede usarse el operador [] para crear una nueva clave, asignándole un valor, o cambiar el valor de una clave que ya ha sido generada. Debemos tener sumo cuidado al usar este método para lectura, ya que si la clave especificada no existe, la creará asignándole un valor vacío.

Finalmente, se recorre la estructura mediante un iterador y se muestran todos los datos accediendo a los miembros `first` y `second` de cada par contenido.

2.2. Iteradores

Iteradores
Son objetos cuyo comportamiento es similar al de un puntero. Nos permiten movernos dentro de los contenedores y hacer referencia a una posición de memoria.

Hasta aquí hemos visto gran parte de la funcionalidad de la librería STL, estudiando algunos de sus contenedores y tomando contacto con el concepto y la utilización de iteradores. Ahora, profundizaremos en estos últimos para poder, en el parágrafo 2.3, comprender la utilización del tercer componente de la librería: los algoritmos.

Dijimos que un iterador es un objeto que apunta a una posición dentro de un contenedor y nos permite movernos dentro del mismo. Además, pudimos observar que cada contenedor define su propio tipo de iterador internamente; por ejemplo, la declaración `vector<int>::iterator p;` significa que *p* es del tipo iterador que fue definido dentro de la clase templatizada vector, utilizada con enteros.

Dado que cada contenedor maneja la memoria de forma diferente a los demás, es de esperar que sus iteradores se comporten de forma distinta y permitan diferentes tipos de funcionalidad.

STL define cinco categorías diferentes de iteradores, según se detallan en la siguiente figura:

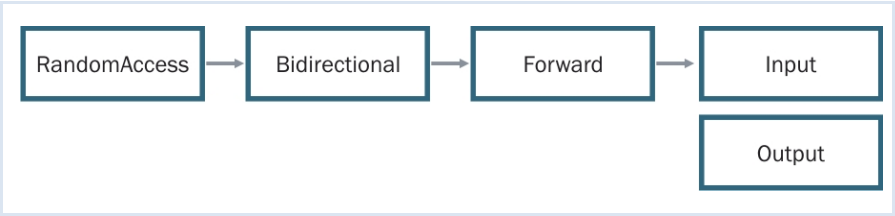

Fig 2. Tipos de iteradores definidos en STL.

Los iteradores de tipo *Input* y *Output* son los más limitados y solamente permiten realizar operaciones de entrada o salida en una posición, es decir, leer o escribir el valor apuntado.

Los iteradores de tipo *Forward* tienen la funcionalidad de los iteradores Input y Output, pero permiten, además, moverse dentro del contenedor en un solo sentido.

Los iteradores *bidireccionales* son similares a los anteriores, pero pueden desplazarse en ambas direcciones. Es el caso de los iteradores definidos para el contenedor list, ya que se trata de una lista doblemente enlazada.

Los iteradores de acceso aleatorio o *RandomAccessIterator* son los más generales y permiten realizar sobre él cualquier operación que se haga con un puntero. Implementan todas las funcionalidades de los iteradores anteriores y también permiten realizar accesos no secuenciales. Esto significa que pueden moverse a cualquier lugar del contenedor de manera inmediata (los iteradores anteriores son secuenciales). El iterador definido en la clase vector es de este tipo.

2.3 Algoritmos

Algoritmos
Son funciones genéricas que nos permiten procesar los datos guardados en los contenedores. Proveen algunas funcionalidades como inicializar, buscar, ordenar y copiar datos de los contenedores.

El último componente que analizaremos es un conjunto de algoritmos genéricos, los cuales proveen diversas funcionalidades y pueden ser utilizados en la mayoría de los contenedores.

Los algoritmos reciben iteradores como parámetros, atendiendo a determinados requerimientos Por ejemplo, el algoritmo `sort()` puede ordenar cualquier contenedor cuyos iteradores sean de tipo RandomAccess, por lo cual será posible utilizar el algoritmo con un vector, pero no con un contenedor list. Por ello, el contenedor implementa su propio algoritmo de ordenamiento como una función miembro.

La generalidad de los algoritmos permite que su utilidad no se limite sólo a contenedores de la STL. Dado que un puntero es un iterador de acceso aleatorio, los algoritmos pueden usarse también con cualquier arreglo estático.

Los algoritmos están definidos en la cabecera <algorithm>. A continuación, describiremos brevemente aquellos que nos serán de mayor utilidad y mostraremos un ejemplo de su utilización.

```
template <class InputIterator, class T>
InputIterator find(     InputIterator first,
                        InputIterator last,
                        const T& value );
```

Realiza una búsqueda del valor *value* en el rango *[first, last).* Devuelve un iterador a la posición de la primera ocurrencia de *value,* o el iterador *last,* en caso de no encontrar el valor.

```
template <class InputIterator, class T>
unsigned count( InputIterator first,
                InputIterator last,
                const T& value );
```

Devuelve cuantos valores iguales a value hay en el rango [first, last).

```
template <class ForwardIterator, class T>
ForwardIterator remove( ForwardIterator first,
                        ForwardIterator last,
                        const T& value );
```

Elimina todos los valores iguales a *value* comprendidos en el rango *[first, last).* Devuelve un iterador al final del nuevo rango (la eliminación hará que el puntero last ya no apunte al final).

template <class ForwardIterator, class T>

```
void fill(      ForwardIterator first,
                ForwardIterator last,
                const T& value );
```

Rellena todos los elementos en el rango *[first, last)* con el valor *value.*

```
template <class ForwardIterator>
ForwardIterator unique (ForwardIterator first,
                        ForwardIterator last );
```

Elimina los elementos repetidos, dejando sólo la primera ocurrencia de cada valor. Para ganar eficiencia, no elimina realmente los valores, sino que los mueve al final de la estructura. Devuelve un iterador al final del nuevo rango (o al principio de donde se encuentran los valores repetidos). Funciona únicamente con contenedores cuyos elementos están ordenados.

```
template <class RandomAccessIterator>
void random_shuffle(    RandomAccessIterator first,
                        RandomAccessIterator last );
```

Mezcla los valores del contenedor en el rango *[first, last).* Sólo funciona con iteradores de acceso aleatorio.

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

Ordena los valores del contenedor en el rango *[first, last).* Solamente funciona con iteradores de acceso aleatorio.

```
template <class ForwardIterator, class T>
bool binary_search(     ForwardIterator first,
                        ForwardIterator last,
                        const T& value );
```

Realiza una búsqueda binaria del valor *value* en el rango *[first, last).* Devuelve *verdadero* si lo encuentra. Los valores del contenedor deben estar, por supuesto, ordenados.

```
template <class ForwardIterator>
ForwardIterator max_element(    ForwardIterator first,
                                ForwardIterator last );
```

Devuelve un iterador a la posición del mayor elemento contenido en el rango *[first, last).*

```
template <class ForwardIterator>
ForwardIterator min_element(    ForwardIterator first,
                                ForwardIterator last );
```

Devuelve un iterador a la posición del menor elemento contenido en el rango *[first, last).*

```
template <class InputIterator, class Function>
Function for_each(      InputIterator first,
                        InputIterator last,
                        Function f );
```

Llama a la función *f* para cada elemento en el rango *[first, last).*

```
template<       class InputIterator1,
                class InputIterator2,
                class OutputIterator>
OutputIterator merge(   InputIterator1 first1,
```

```
                                InputIterator1 last1,
                                InputIterator2 first2,
                                InputIterator2 last2,
                                OutputIterator result );
```

Combina los elementos ordenados en los rangos *[first1, last1)* y *[first2, last2).* El resultado es insertado en el contenedor y posición apuntados por result.

El siguiente código ejemplifica el uso de algunos de estos algoritmos con un vector:

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main(int argc, char *argv[]) {
    vector<int> v;
    for(unsigned i=0; i<20; i++) v.insert(v.end(), rand()%11);

    cout<<"v=";
    for(unsigned i=0; i<20; i++) cout<<v[i]<<" ";
    cout<<endl;

    // ordenamos la segunda mitad del vector
    cout<<"Ordenando la segunda mitad del vector"<<endl;
    vector<int>::iterator mitad=v.begin()+(v.size()/2);
    sort(mitad, v.end());
    cout<<"v=";
    for(unsigned i=0; i<v.size(); i++) cout<<v[i]<<" ";
    cout<<endl;

    // eliminamos los repetidos de la segunda mitad
    cout<<"Eliminando los repetidos de la segunda"<<endl;
    vector<int>::iterator nuevofinal;
    nuevofinal=unique(mitad, v.end());
    v.erase(nuevofinal, v.end());
    cout<<"v=";
    for(unsigned i=0; i<v.size(); i++) cout<<v[i]<<" ";
    cout<<endl;

    // buscamos el mayor y el menor de la primer mitad
    cout<<"El mayor de la primer mitad: ";
    cout<<*max_element(v.begin(), mitad)<<endl;
    cout<<"El menor de la primer mitad: ";
    cout<<*min_element(v.begin(), mitad)<<endl;


    return 0;
}

```

Para obtener mayor información sobre todos los contenedores de STL, se puede consultar en http://www.cplusplus.com/reference/stl/. Asimismo, para contar con una referencia completa de los algoritmos, sugerimos ingresar a http://www.cplusplus.com/reference/algorithm/.

En 2010, Electronic Arts® liberó el código fuente de su propia versión de la librería STL, llamada EASTL. La librería fue desarrollada, por supuesto, con el propósito de satisfacer las exigencias extremas del uso de la memoria y CPU en el ámbito de los juegos. Para más información, consultar en http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.htm

3. String

Si bien esta clase no forma parte de la STL, su estructura y comportamiento es muy similar a la de los contenedores que hemos visto. Por lo tanto, aprovecharemos esta oportunidad para hacer una pequeña revisión de sus funcionalidades.

String
Es un objeto que permite manipular cadenas de caracteres dinámicas de forma sencilla y cómoda para el usuario.

Los objetos de tipo *string* permiten manipular cadenas de caracteres dinámicas de forma sencilla y cómoda para el usuario.

Veamos, como lo hicimos con los contenedores de STL, una breve descripción de sus métodos y un ejemplo de su utilización. La definición de la clase está dentro de la cabecera <iostream>

```
string ( );
string ( const string& str );
string ( const char * ) ;
```

La clase string admite numerosos constructores. Algunos de ellos permiten inicializar el objeto desde otro string, a partir de un cstring (un arreglo de caracteres terminado en '\0').

```
size_t size();
size_t length();
```

Devuelven la longitud de la cadena, es decir, la cantidad de caracteres que posee.

```
void clear();
```

Limpia la cadena, igualándola a una cadena vacía.

```
bool empty();
```

Devuelve *verdadero* si la cadena no posee ningún carácter.

```
string& operator=( const string& str );
string& operator=( const char* s );
string& operator=( char c );
```

Permite asignar a la cadena otra string, cstring, o un único carácter.

```
char& operator[]( size_t pos );
```

Devuelve una referencia al carácter en la posición *pos,* permitiendo tanto su lectura como escritura. No asegura que *pos* sea una posición válida, por lo que será responsabilidad del programador mantenerse dentro del rango válido.

```
string& operator+( const string& str );
string& operator+( const char* s );
string& operator+( char c );
```

Produce una nueva cadena con la concatenación de la string que realizó la llamada y otra string, cstring o carácter.

```
string& operator+=( const string& str );
string& operator+=( const char* s );
string& operator+=( char c );
```

Permite concatenar un string, cstring o carácter a la cadena actual.

```
string substr ( size_t pos = 0, size_t n = npos )
```

Devuelve la subcadena de *n* caracteres, a partir de la posición pos.

```
string& replace ( size_t pos1, size_t n1, const string& str );
string& replace (       string::iterator i1,
                        string::iterator i2,
                        const string& str );
```

Permite remplazar una porción de la cadena *str*. La primera versión recibe la posición de inicio de la subcadena a reemplazar y su tamaño. La segunda versión recibe los iteradores que delimitan la subcadena a reemplazar.

```
size_t find ( const string& str, size_t pos = 0 );
```

Devuelve la posición con la primera ocurrencia de la cadena str, a partir de la posición pos. En caso de no haber ninguna ocurrencia, devuelve la constante `string::npos`. También existen sobrecargas para buscar cstrings y caracteres.

```
size_t rfind ( const string& str, size_t pos = 0 );
```

Su funcionamiento es similar al de `find()`, pero comenzando desde el final de la cadena hacia atrás. Devuelve la posición con la última ocurrencia de la cadena str, entre el principio de la cadena y la posición pos. Devuelve la constante string::npos en caso de no haber ninguna ocurrencia. También existen sobrecargas para buscar cstrings y caracteres.

```
const char* c_str();
```

Devuelve un cstring con el contenido de la cadena, lo cual permite utilizar objetos string en funciones que requieren un cstring.

Veamos un ejemplo sobre la utilización de esta clase:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
    string cadena=
    "Casi todos creen que el tiempo es como un rio \
que fluye seguro y veloz en una sola direccion, \
pero yo le he visto la cara al tiempo y les puedo \
asegurar que estan equivocados. El tiempo es un \
oceano en la tormenta.";

    // mostramos la cadena
    cout<<cadena<<endl<<endl;

    // contamos la cantidad de veces que aparece
    // la palabra tiempo
    int ntiempo=0, pos;
    pos=cadena.find("tiempo");
    while(pos!=string::npos){
        ntiempo++;
        // busca "tiempo" empezando desde pos+6
        pos=cadena.find("tiempo", pos+6);
    }
    cout<<"La palabra tiempo aparece "<<ntiempo;
    cout<<" veces en la cadena"<<endl<<endl;

    // realizamos algunas modificaciones
    cadena.replace(0, 10, "Ustedes");
    cout<<cadena<<endl<<endl;

    // mostramos solo una porcion de la cadena
    cout<<cadena.substr(173, 22)<<endl;

    return 0;
}

```

Bibliografía

Balagurusamy, E. *Object Oriented Programing With C++*. Tata McGraw Hill Publishing Company Limited, 2008, 4th ed.

[cplusplus.com] http://www.cplusplus.com/reference [21,07,11]

Stroustrup, Bjarne. *The C++ Programming Language.* Addison Wesley Longman, 1997, 3° ed.

[wikipedia] http://en.wikipedia.org