Contenedores

¿Que es?

Un *contenedor* de la **STL** es una clase genérica que puede instanciarse para representar diversos tipos de objetos.

Esta clase incluye ciertas operaciones sobre los objetos de su tipo. Naturalmente estas operaciones están representadas por funciones-miembro, incluyendo constructores y funciones-operador, que son a su vez funciones genéricas.

Una estructura de datos se dice que es contenedora si puede contener instancias de otras estructuras de datos. En concreto, la STL dispone de las siguientes estructuras:

- Secuencias
    - deque
    - list
    - stack
    - vector
- adaptadores
    - queue
- Contenedores asociativos
    - map
    - multimap
    - set
    - multiset

Los contenedores se dividen en tres categorías:

- **Contenedores lineales**. Almacenan los objetos de forma secuencial permitiendo el acceso a los mismos de forma secuencial y/o aleatoria en función de la naturaleza del contenedor.
- **Contenedores asociativos**. En este caso cada objeto almacenado en el contenedor tiene asociada una clave. Mediante la clave los objetos se pueden almacenar en el contenedor, o recuperar del mismo, de forma rápida.
- **Contenedores adaptados**. Permiten cambiar un contenedor en un nuevo contenedor modificando la interface (métodos públicos y datos miembro) del primero. En la mayor parte de los casos, el nuevo contenedor únicamente requerirá un subconjunto de las capacidades que proporciona el contenedor original.

La adopción de STL ofrece varias ventajas:

- Al ser estándar, está (o estará) disponible por todos los compiladores y plataformas. Esto permitirá utilizar la misma librería en todos los proyectos y disminuirá el tiempo de aprendizaje necesario para que los programadores cambien de proyecto.
- El uso de una librería de componentes reutilizables incrementa la productividad ya que los programadores no tienen que escribir sus propios algoritmos. Además, utilizar una librería libre de errores no sólo reduce el tiempo de desarrollo, sino que también incrementa la robustez de la aplicación en la que se utiliza.

- Las aplicaciones pueden escribirse rápidamente ya que se construyen a partir de algoritmos eficientes y los programadores pueden seleccionar el algoritmo más rápido para una situación dada.
- Se incrementa la legibilidad del código, lo que hará que éste sea mucho más fácil de mantener.
- Proporciona su propia gestión de memoria. El almacenamiento de memoria es automático y portátil, así el programador ignorará problemas tales como las limitaciones del modelo de memoria del PC.

vector

Este contenedor es una estructura de datos de tamaño fijo preferentemente. Aunque la STL proporciona herramientas para cambiar el tamaño de un vector de forma dinámica, esta operación es costosa y debe ser evitada dentro de lo posible.

Disponen de un buén mecanismo de acceso aleatorio a elementos y de un mecanismo de inserción al final muy eficiente.

Generalmente es preferible utilizar un vector que un deque o un list, a menos que sea frecuentemente necesario insertar datos al comienzo o al final, en cuyo caso es mejor utilizar un deque. Si por el contrario es frecuente la inserción de elementos en el centro, entonces es preferible un list.

Uso

Podemos declarar vectores de cualquier tipo, el vector puede estar vacío o puede tener un tamaño.

```
#include <vector>
vector<double> scores(20);  //Decaracion de vector con tamaño definido
vector<string> names;       ////Decaracion de vector sin tamaño definido
vector<bool> busyFlags(5);
vector<Student> classRoll(50);
```

```
#include <vector>
int main(int argc, char *argv[]) {
    char buffer[80];
    double suma;
    /* Definicion de vector de valores 'double' */
    vector<double> v;

    v.push_back(999.25);    // Agrega valor al fnal de vector
    v.push_back(888.50);
    v.push_back(777.25);

    suma = 0;
    for(int i = 0; i < v.size(); i++){
        suma += v[i];   // operador[] para acceder al indice
        printf(buffer, "%10.2f", v[i]);
        cout << buffer << endl;
    }

    cout << "----------" << endl;
    printf(buffer, "%10.2f", suma);
    cout << buffer << endl;
    cin.get();
    return 0;
}
```

list

Este contenedor responde a la idea intuitiva de “lista”, el almacenamiento de objetos en una secuencia lineal que no está necesariamente ordenada. Estos contenedores suelen ser implementados como listas doblemente enlazadas.

Dispone de mecanismos eficientes para insertar elementos al principio, al final o en cualquier posición. Estas operaciones consumen un tiempo constante con independencia del número de elmentos albergados en el contenedor.

Dado que son estructuras lineales, en general los elementos no pueden ser accedidos por subíndices como en un vector. Es necesario realizar un recorrido lineal por todos los valores, por lo que en las operaciones de acceso se utilizan tiempos proporcionales al número de elementos.

Uso

```
#include <list>

using namespace std;

int main(int argc, char *argv[]) {
    list<double> lalista;   // Definicion de lista de tipo 'double'
    double num, suma=0;

    cout << "Una sencilla calculadora" << endl;

    do {
        cout << "Ingrese un número, 0 para salir: ";
        cin  >> num;
        if (num != 0) lalista.push_back(num);   // Agrega valor al final de la lista
    } while (num != 0);

    cout << "----------" << endl;

    while( !lalista.empty() ) {
        num = lalista.front();
        cout << setw(10) << num << endl;
        suma += num;
        lalista.pop_front();
    }
    cout << "----------" << endl;
    cout << setw(10) << suma << endl;
    return 0;
}
```

stack

Contenedor de elementos tipo pila LIFO que permite inserciones y eliminaciones solo en la parte superior.

Uso

```cpp
#include <stack>

using namespace std;

int main() {
  stack<int> stack1; // Definicion de stack vacia
  stack1.push(100); // Agrega elemento a la cola
  stack1.push(200);
  stack1.push(300);
  stack1.push(400);
  stack1.push(500);

  /* Valor del primer elemento de la cola */
  cout << "El primer elemento de la cola es: " << stack1.top() << endl;
  /* Tamaño de la cola */
  cout << "El tamaño de la cola es: " << stack1.size() << endl;

  if (stack1.empty()) {
    cout << "La coa esta vacia" << endl;
  } else {
    cout << "la cola no esta vacia" << endl;
  }
}
```

deque

Double-ended queue es un tipo de estructura de datos que comparte las características de las colas *Queues* y las pilas *Stacks*. Como en las colas, los elementos pueden ser empujados por un extremo al interior del contenedor, y el primer elemento introducido puede ser extraído por el extremo opuesto. Al mismo tiempo, el último elemento introducido por el principio puede ser extraído en ese mismo extremo como si fuese una pila.

Estos contenedores suelen ser implementados bajo la forma de matrices bidimensionales.

Las características de los deques implementados en la STL pueden resumirse en: Acceso aleatorio; mecanismo eficiente de inserción al principio o al final.

Uso

```cpp
#include <deque>

using namespace std;

void showdq(deque <int> g) {
    deque <int> :: iterator it;
    for (it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

int main() {
    deque <int> gquiz;  // Definicion de cola de doble terinacion de tipo 'int'
    gquiz.push_back(10);    // Agrega valor al final de la cola
    gquiz.push_front(20);   // Agrega valor al principo de la cola
    gquiz.push_back(30);
    gquiz.push_front(15);
    cout << "La cola de doble terminacion es: ";
    showdq(gquiz);

    cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.max_size() : " << gquiz.max_size();

    cout << "\ngquiz.at(2) : " << gquiz.at(2);
    cout << "\ngquiz.front() : " << gquiz.front();
    cout << "\ngquiz.back() : " << gquiz.back();

    cout << "\ngquiz.pop_front() : ";
    gquiz.pop_front();
    showdq(gquiz);

    cout << "\ngquiz.pop_back() : ";
    gquiz.pop_back();
    showdq(gquiz);

    return 0;
}
```

set

Mantiene los elementos en orden. Dispone de mecanismos eficientes para inclusión, inserción y eliminación de elementos, y soporta claves únicas.

Uso

```
#include <set>
#include <iterator>

using namespace std;

int main() {
    set <int, greater <int> > gquiz1; // Definicion de "set" vacio

    /* Inserta valores en orden aleatorio */
    gquiz1.insert(40);
    gquiz1.insert(30);
    gquiz1.insert(60);
    gquiz1.insert(20);
    gquiz1.insert(50);
    gquiz1.insert(50);
    gquiz1.insert(10);

    /* Muestra dato con uso de iterador */
    set <int, greater <int> > :: iterator itr;
    cout << "\nThe set gquiz1 is : ";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {
        cout << '\t' << *itr;
    }
    cout << endl;
    return 0;
}
```

multiset

Permite la existencia de claves duplicadas. Es decir, distintos elementos dentro del conjunto pueden responder a la misma clave.

Uso

```
#include <set>

using namespace std;

int main() {
    multiset<int, greater<int>> set_1;  // Define "multiset" vacio

    set_1.insert(502);  //  Agrega valores al multiset
    set_1.insert(502);
    set_1.insert(506);
    set_1.insert(507);
    set_1.insert(555);

    /* Iterador para multiset */
    multiset<int, greater<int>>::iterator iterator_1;
    cout << "\nThe multiset elements are: ";
    for (iterator_1 = set_1.begin(); iterator_1 != set_1.end(); ++iterator_1) {
        cout << "\t" << *iterator_1;
    }
}
```

map

Mantiene sus elementos ordenados. Se caracteriza porque sus miembros son pares de valores que pueden ser de tipos distintos. Uno de ellos, el que actúa como clave para el índice, puede ser de cualquier tipo, a condición de que sus elementos puedan ser ordenados según un criterio.

Permite claves únicas. Es decir, que solo puede existir un miembro para cada clave. Dispone de mecanismos de inserción y borrado muy eficientes y no existe límite de tamaño. La estructura se encarga de crecer y disminuir en concordancia con las necesidades de sus miembros. Permite el operador [ ] subíndice para los elementos de la clave así como otras técnicas de acceso.

Uso

```
#include <iostream>
#include <iterator>
#include <map>
using namespace std;

int main() {
    map<int, char> map1;    // Definicion de map vacio
    map<int, char>::iterator cursor;
    map1[1]='a';    //Asigncaion de valor "a" a clave "1"
    map1[2]='b';
    cout <<"KEY\tELEMENT"<<endl;
    for(cursor = map1.begin(); cursor!=map1.end(); cursor++) {
        cout<<cursor->first;    //Acceso a la clave del objeto map
        cout<< '\t'<<cursor->second<<'\n'<<endl;    //Acceso al del objeto map
    }
}
```

multimap

Basado en la estructura map permitiendo además claves duplicadas. Es decir, que una misma clave pueda estar asociada a dos “valores” distintos.

Uso

```cpp
#include <map>

using namespace std;

int main()  {
    multimap<string, int> m;    // Definicion de multimap vacio

    m.insert(pair<string, int>("a", 1));    // Agrega valor a clave "a"
    m.insert(pair<string, int>("c", 2));    // Agrega valor a clave "c"
    m.insert(pair<string, int>("b", 3));    // Agrega valor a clave "b"
    m.insert(pair<string, int>("b", 4));    // Agrega otro valor a clave "b"
    m.insert(pair<string, int>("a", 5));    // Agrega otro valor a clave "b"
    m.insert(pair<string, int>("b", 6));

    cout << "Number of elements with key a: " << m.count("a") << endl;
    cout << "Number of elements with key b: " << m.count("b") << endl;
    cout << "Number of elements with key c: " << m.count("c") << endl;

    cout << "Elements in m: " << endl;
    for (multimap<string, int>::iterator it = m.begin(); it != m.end(); ++it) {
        cout << "  [" << (*it).first << ", " << (*it).second << "]" << endl;
    }

    pair<multimap<string, int>::iterator, multimap<string, int>::iterator> ppp;
    ppp = m.equal_range("b");
    cout << endl << "Rango de elementos de \"b\":" << endl;
    for (multimap<string, int>::iterator it2 = ppp.first; it2 != ppp.second;++it2) {
        cout << "  [" << (*it2).first << ", " << (*it2).second << "]" << endl;
    }
    m.clear();
    return 0;
}
```

queue

Contenedor tipo cola FIFO que permite inserciones al final y eliminaciones al principio.

Uso

```
#include <queue>

using namespace std;

int main() {

    queue<int> ourQueue;    // Definicion de cola vacia

    cout<<"Espacio ocupado en la memoria "<<ourQueue.size()<<endl;
    ourQueue.emplace(3);
    ourQueue.emplace(6);
    ourQueue.emplace(7);
    while( ! ourQueue.empty() ) {
            int iTemp = ourQueue.front();
            cout<<iTemp<<endl;
            ourQueue.pop();
    }
    return 0;
}
```