

Biblioteca estándar de templates de C++ Standard Template Library (STL)

- Tipos de Contenedores en STL
- Iteradores en STL
- STL: contenedores list, set y map
- Algoritmos genéricos
- Invalidación de iteradores

STL de C++

- Los componentes que provee son:
 - clases de contenedores;
 - iteradores para cada tipo de contenedor;
 - algoritmos para manipular los datos guardados en los contenedores.
- STL hace un uso intensivo del mecanismo de parametrización.
- La documentación de la STL describe cada conjunto de requerimientos que cumplen los tipos de datos que provee a través de lo que denomina “conceptos”.
- De esta forma, partiendo de conceptos muy generales que abarcan a todos los contenedores (o todos los iteradores), los mismos se van refinando en forma sucesiva hasta que llegan a conceptos muy especializados y que especifican a un solo tipo de datos.

Por ejemplo: contenedor --> secuencia --> vector

STL: Tipos de contenedores

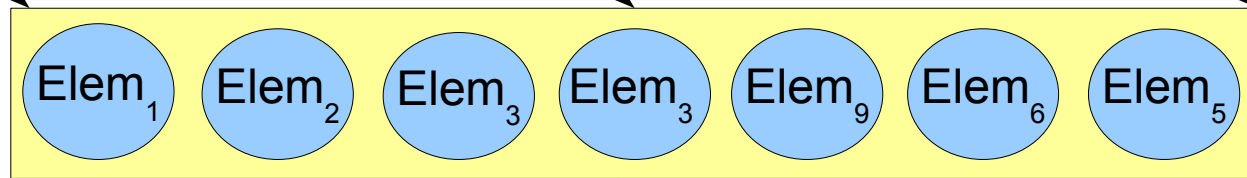
- De todos los conceptos, los más importantes son aquellos que dividen a los contenedores en dos clases principales:
 - **Secuenciales**: los elementos se encuentran en un orden lineal estricto, pudiéndose insertar y eliminar elementos de posiciones específicas: **vector**, **deque**, **list**, **slist**
 - **Asociativos**: permiten manipular a los elementos a través de una clave (la cual una vez almacenada no puede modificarse). La característica más importante es que las operaciones de acceso utilizando las claves son eficientes ($O(\log(n))$). Los mismos a su vez se dividen en:
 - **Asociativos simples**: La clave es el mismo elemento que se almacena. Esto lleva a que los elementos guardados no puedan modificarse: **set**, **multiset**, (**hash_set**, **hash_multiset**)
 - **Asociativos por pares**: La clave y el elemento guardado son objetos distintos, por lo que se pueden modificar los elementos: **map**, **multimap**, (**hash_map**, **hash_multimap**)

STL: Secuencias

Inserción al principio

Inserción en una posición intermedia

Inserción al final

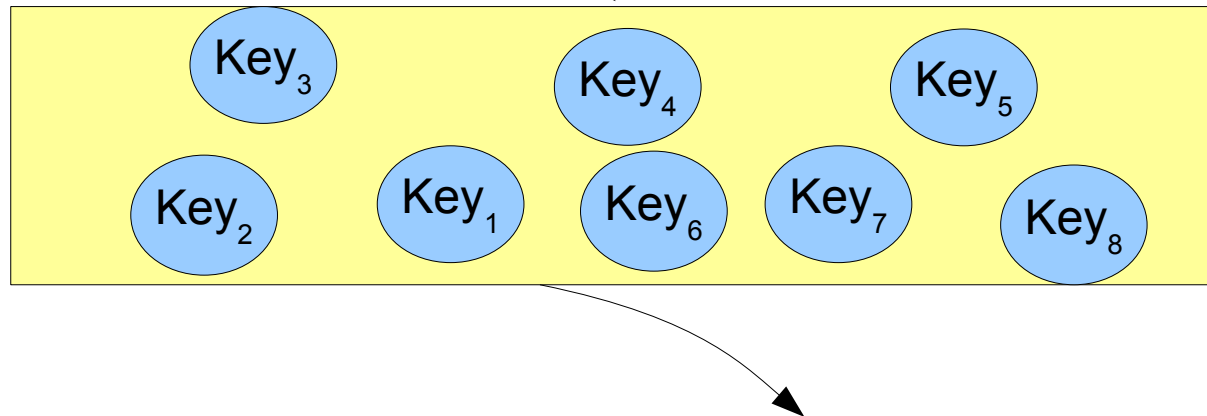


No soporta la función de búsqueda. Si queremos consultar un elemento determinado, debemos iterar sobre toda la colección de elementos.

- Las inserciones y eliminaciones en la list tienen costo $O(1)$.
- La inserción y eliminación en el final en el vector tiene costo $O(1)$, el resto $O(n)$.
- La búsqueda de un elemento tiene costo $O(n)$ (la debemos implementar nosotros).

STL: Asociaciones simples

Inserción (ordenada) respecto a una comparación definida sobre el elemento

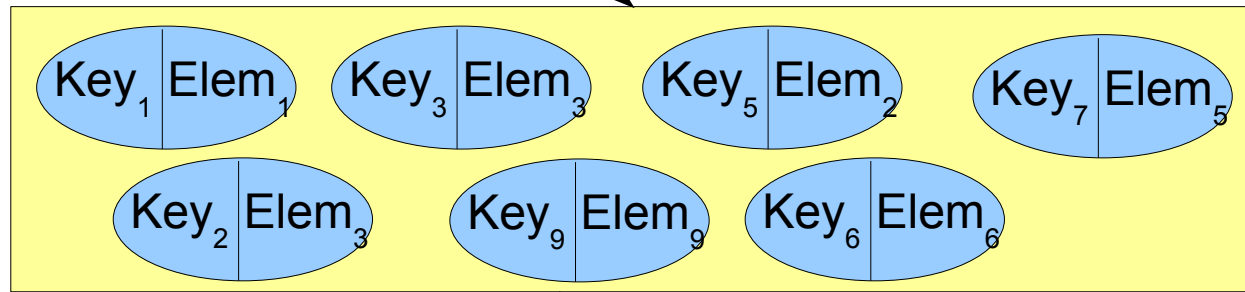


Se brinda soporte para accesos eficientes a partir del valor del elemento.

- Las inserciones y eliminaciones en el (multi)set tienen costo $O(\log(n))$.
- La búsqueda de un elemento tiene costo $O(\log(n))$ (siempre que utilicemos el mismo criterio de comparación que se usó para la inserción).
- No se permite la modificación de los elementos guardados.

STL: Asociaciones por pares

Inserción (ordenada) del par **[Clave,Elemento]**, respecto a una comparación definida sobre la clave



Se brinda soporte para accesos eficientes a los *elementos* a partir del valor de la *clave*

- Las inserciones y eliminaciones en el (multi)map tienen costo $O(\log(n))$.
- La búsqueda de un elemento tiene costo $O(\log(n))$ (siempre que utilicemos el mismo criterio de comparación que se usó para la inserción).
- Se permite la modificación de los elementos guardados (pero no la clave).

STL – Iteradores: Interfaz general (1)

- Definición
 - *TipoContenedor::iterator nombrelterador;*
- Utilización
 - Avanzar iterador al siguiente elemento:
it++
 - Acceder al valor de un elemento:
***it**
 - Cuando los elementos almacenados son instancias de alguna clase, también es válido:
 - **it->atributo**
 - **it->método()**

STL – Iteradores: Interfaz general (2)

- Inicializar iterador:

```
it = contenedor.begin();
```

- Consultar si un iterador llegó al final de la secuencia:

```
if (it != contenedor.end()) {  
    // El iterador referencia un elemento válido  
} else {  
    // El iterador llegó al final  
}
```


STL – Contenedores: list

- Declaración

```
#include "list"
using namespace std;
list<TipoDato> lista;
list<TipoDato>::iterator iterador;
```

- Operaciones

- Modificadoras:

- Agregar al principio: **lista.push_front(dato)**
 - Agregar al final: **lista.push_back(dato)**
 - Agregar antes de una posición: **lista.insert(iterador, dato)**
 - Eliminar el elemento en una posición: **lista.erase(iterador)**
 - Eliminar un elemento: **lista.remove(dato)**

STL – Contenedores: list

- Operaciones
 - Consultoras y búsquedas:
 - Sacar del principio: **lista.pop_front()**
 - Sacar del final: **lista.pop_back()**
 - Obtener un elemento particular: ... **no existe una función de soporte para búsquedas. Hay que iterar sobre los elementos.**
 - Además las lista tienen funciones para: vaciar, intercambiar, ordenar, fusionar, comparar, eliminar elementos repetidos, insertar rangos de elementos, etc.

List: Ordenamiento por selección

```
list<int> enteros;  
// ...  
list<int>::iterator menor = enteros.begin();  
while (menor != enteros.end()) {  
    list<int>::iterator it = menor;  
    while (it != enteros.end()) {  
        if (*it < *menor) {  
            int temp = *it;  
            *it = *menor;  
            *menor = temp;  
        }  
        it++;  
    }  
    menor++;  
}
```

STL – Contenedores: set

- Declaración

```
#include "set"
using namespace std;
set<TipoDato> conjunto;
set<TipoDato>::iterator iterador;
```

- Operaciones

- Modificadoras:

- Agregar un elemento: **conjunto.insert(dato)**
 - Eliminar un elemento: **conjunto.erase(dato)**
 - Eliminar el elemento en una posición:
conjunto.erase(iterador)

STL - Contenedores: set

- Operaciones
 - Consultoras y búsquedas:
 - Obtener una referencia a la posición de un elemento:
iterador = conjunto.find(dato)
 - Además el tipo set tiene funciones para: vaciar, intercambiar, comparar, insertar rangos de elementos, búsqueda por rangos, etc.
- El tipo multiset es la versión del set que soporta elementos/claves repetidos.

STL: Comparación list y set (1)

```
class Punto {
public:
    Punto(int x, int y) { this->x = x; this->y = y; }
    int getX() const { return x; }
    int getY() const { return y; }
    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }
    bool operator < (const Punto & p) const {
        if (x < p.x) { return true; }
        else if (x > p.x) { return false; }
        else { return y < p.y; } }

private:
    int x, y;
};

ostream & operator << (ostream & stream, const Punto & punto) {
    return stream << "(" << punto.getX() << "," << punto.getY()
    << ")";
}
```

STL: Comparación list y set (2)

```
set<Punto> puntos;
```

```
puntos.insert(Punto(3,5));  
puntos.insert(Punto(1,1));  
puntos.insert(Punto(1,3));  
puntos.insert(Punto(3,4));  
puntos.insert(Punto(3,4));  
puntos.insert(Punto(1,1));
```

```
set<Punto>::iterator it = puntos.begin();  
while (it != puntos.end()) {  
    cout << *it << "\n";  
    it++;  
}
```

STL: Comparación list y set (3)

```
list<Punto> puntos;
```

```
puntos.push_front(Punto(3,5));  
puntos.push_front(Punto(1,1));  
puntos.push_front(Punto(1,3));  
puntos.push_front(Punto(3,4));  
puntos.push_front(Punto(3,4));  
puntos.push_front(Punto(1,1));
```

```
puntos.sort();
```

```
list<Punto>::iterator it = puntos.begin();  
while (it != puntos.end()) {  
    cout << *it << "\n";  
    it++;  
}
```


STL – set: Búsquedas

```
int x,y;
cin>>x;
cin>>y;

Punto p(x,y);

set<Punto>::iterator it = puntos.find(p);
if (it != puntos.end()) {
    cout << "Punto Encontrado: " << *it;
} else {
    cout << "No se encontró el punto: " << p;
}
```

STL – Contenedores: map

- Declaración:

```
#include "map"
using namespace std;
map<TipoClave, TipoDato> mapa;
map<TipoClave, TipoDato>::iterator iterador;
```

- Operaciones

- Modificadoras:

- Agregar un elemento: **mapa[clave] = dato**
 - Eliminar un elemento: **mapa.erase(clave)**
 - Eliminar el elemento en una posición: **mapa.erase(iterador)**

STL – Contenedores: map

- Operaciones
 - Consultoras y búsquedas:
 - Obtener una referencia a la posición de un elemento:
iterador = mapa.find(clave)
*(tener en cuenta que los elementos guardados **son pares** [clave,dato], por lo tanto los iteradores son referencias a los mismos)*
 - Además el tipo map tiene funciones para: vaciar, intercambiar, comparar, insertar rangos de elementos, búsqueda por rangos, etc.
- El tipo multimap es la versión del map que soporta claves repetidas.

STL – Contenedores: map

```
class Punto {
public:
    Punto(int x, int y) { this->x = x; this->y = y; }
    int getX() const { return x; }
    int getY() const { return y; }
    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }

private:
    int x, y;
};

ostream & operator << (ostream & stream, const Punto & punto) {
    return stream << "(" << punto.getX() << "," << punto.getY()
    << ")";
}

class ComparadorPunto {
public:
    bool operator () (const Punto & p1, const Punto & p2) const {
        if (p1.getX() < p2.getX()) { return true; }
        else if (p1.getX() > p2.getX()) { return false; }
        else { return p1.getY() < p2.getY(); }
    }
};
```

STL – Contenedores: map

```
map<Punto, list<int>, ComparadorPunto> elementosPorPunto;
```

```
list<int> lista1;
```

```
lista1.push_back(1);
```

```
elementosPorPunto[Punto(1,1)] = lista1;
```

```
elementosPorPunto[Punto(1,2)] = lista1;
```

```
elementosPorPunto[Punto(1,2)].push_back(2);
```

```
lista1.push_back(3);
```

```
elementosPorPunto[Punto(1,3)] = lista1;
```

```
map<Punto, list<int>, ComparadorPunto>::iterator it =  
    elementosPorPunto.begin();
```

```
while (it != elementosPorPunto.end()) {  
    cout << it->first << "\n";  
    mostrarLista(it->second);  
    it++;  
}
```

STL – map: Búsquedas

```
int x,y;
cin>>x;
cin>>y;

Punto p(x,y);

map<Punto,list<int>,ComparadorPunto>::iterator it =
    elementosPorPunto.find(p);
if (it != elementosPorPunto.end()) {
    cout << "Punto Encontrado: " << it->first << "\n";
    mostrarLista(it->second);
} else {
    cout << "No se encontró el punto: " << p;
}
```

STL: Algoritmos genéricos

```
template<typename InputIterator>
void imprimirElementos(InputIterator inicio, InputIterator fin) {
    InputIterator it = inicio;
    while (it != fin) {
        cout << *it << "\n";
        it++;
    }
}
```

// Ahora no importa si almacenamos los puntos en un list o en un set.

```
imprimirElementos(puntos.begin(), puntos.end());
```

STL: Invalidación de iteradores (1)

- vector:
 - Cambio en la capacidad.
 - Tanto la inserción como el borrado de un elemento invalida los iteradores a partir de ese elemento.
- list / set / map:
 - Las inserciones no invalidan iteradores.
 - Las eliminaciones no invalidan los iteradores en otras posiciones. Sólo se invalidan los que referencian al elemento eliminado.

STL: Invalidación de iteradores (2)

- Ejemplo de eliminación sin invalidación:

```
map<Punto, list<int>, ComparadorPunto>::iterator it =
    elementosPorPunto.begin();
map<Punto, list<int>, ComparadorPunto>::iterator itToDel;

while (it != elementosPorPunto.end()) {
    if (it->second.size() == 0)
    {
        itToDel = it;
        it++;
        elementosPorPunto.erase(itToDel);
    } else {
        it++;
    }
}
```

Enlaces

- Standard Template Library (STL) C++
 - Documentación (se puede descargar)
 - <http://www.sgi.com/tech/stl>
- cplusplus.com
 - <http://www.cplusplus.com/reference/stl/>

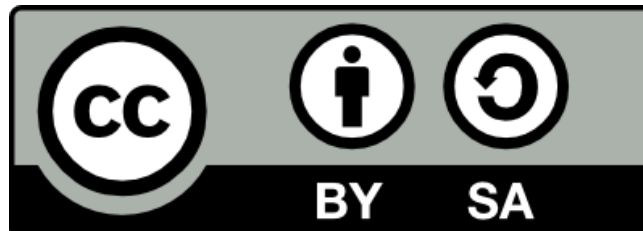
Federico Améndola

Martín Fernández

Consultas: **laboratorio.ayda@alumnos.exa.unicen.edu.ar**

Licencia creative commons

Atribución-Compartir Obras Derivadas Igual 2.5 Argentina



<http://creativecommons.org/licenses/by-sa/2.5/ar/>