



STL

---

John Lambert  
[jlambert@jlambert.com](mailto:jlambert@jlambert.com)



# Overview

---

- Motivación
- ¿Qué es la STL?
- ¿De dónde salió?
- Containers
- Iterators
- Algorithms
- Al fin terminó ...



# ¿Por qué aprender STL?

---

Every line of code you **don't** write  
is bug-free.

Cada línea de código que no  
escribe está libre de errores.



# Perspective

---

- Iremos revisando la mayoría de la STL
  - Al se una biblioteca abierta siempre se puede mirar la sintaxis
  - Se indicarán y explicarán los puntos que pudiesen resultar confusos.
- Comprenderemos facilmente el comportamiento de distintas aplicaciones
- Pasaremos por alto características avanzadas
  - Hecharemos una mirada a programación genérica
  - Fijaremos la mirada a aspectos útiles
- Comencemos ...



# Ejemplo 1

---

- Lea una cantidad arbitraria de valores enteros ( $n \geq 1$ ) del archivo "numbers.txt" y muestre:
  - Mínimo, máximo
  - Media
  - Promedio
  - Media geométrica  $((y_1 * y_2 * \dots * y_n)^{(1/n)})$
- En cuántas líneas de código piensa se puede hacer?
  - Leer, almacenar, ordenar, bucles ...



# Example 1: STL solution

---

```
vector<int> v;
```

```
copy(istream_iterator<int>(ifstream("numbers.txt")),  
     istream_iterator<int>(), back_inserter(v));
```

```
sort(v.begin(), v.end());
```

```
cout << "min/max: " << v.front() << " " << v.back() << endl;
```

```
cout << "median : " << *(v.begin() + (v.size()/2)) << endl;
```

```
cout << "average: " << accumulate(v.begin(), v.end(), 0.0)  
    / v.size() << endl;
```

```
cout << "geomean: " <<  
    pow(accumulate(v.begin(), v.end(), 1.0, multiplies<double>()),  
        1.0/v.size()) << endl;
```



## Ejemplo 2

---

- Escriba un programa que muestre las palabras y la cantidad de veces que aparecen dentro de un archivo. La salida debe mostrarse ordenada por palabras.
  - Lectura de archivos
  - Almacenar en un vector y otro para la cantidad de apariciones (funciones hash) ...



# Ejemplo 2: STL solution

---

```
vector<string> v;
```

```
map<string, int> m;
```

```
copy(istream_iterator<string>(ifstream("words.txt")),  
      istream_iterator<string>(), back_inserter(v));
```

```
for (vector<string>::iterator vi = v.begin();  
     vi != v.end(); ++vi)  
    ++m[*vi];
```

```
for (map<string, int>::iterator mi = m.begin();  
     mi != m.end(); ++mi)  
    cout << mi->first << ": " << mi->second << endl;
```





# ¿Qué es la STL?

- “Standard Template Library”
- Para implementar la biblioteca en forma tradicional:
  - $N$  tipo de datos,  $M$  contenedores distintos y  $K$  algoritmos
  - Posiblemente requeriría  $N * M * K$  implementaciones
    - CountIntegerInList(IntList il, int toFind), CountIntegerInSet, CountDoubleInList, etc.
  - STL (con templates C++):  $N + M + K$  implementaciones
    - Los algoritmos operan sobre contenedores de diferentes tipos
    - `set<int> mySet;`  
`count(mySet.begin(), mySet.end(), 4);`
    - `list<double> myList;`  
`count(myList.begin(), myList.end(), 3.14);`



# Plataformas

---

- STL es parte del Standard C++
- En Visual C++/Studio 6.0
  - Faltan algunas cosas: hash\_map
- In Visual Studio.NET / VC++ 7.0
  - Tiene algunas cosas
- G++ 3.0
- Stlport.org
  - Implementación libre de std C++ (incluye iostreams), con buena relación prestaciones/performance



# STL resumen

---

- Esencialmente, la STL define *algorithms* que operan sobre un *range* en un *container*
- El orden es:
  - Containers
  - Iterators (ranges)
  - Algorithms



# Containers - Contenedores

---

- Lists

- vector, list, deque

- Adaptors

- queue, priority\_queue, stack

- Associative

- map, multimap, set, multiset
- hash\_{above}



# vector<T>

---

- #include <vector>
- Un arreglo dinámica: con acceso al azar, y capacidad para crecer.
- Utiliza la sintaxis de direccionamiento por índice de los arreglos:
  - `vector<int> v(10); v[0] = 4;`



# vector<T>

---

## ■ Constructores

- `vector<T>()`
- `vector<T>(size_t num_elements)`
- `vector<T>(size_t num, T init)`

## ■ Atributos

- `v.empty()` // true if v has 0 elements
- `v.size()` // number of elements
- `v.capacity()` // number of elements v can hold before allocating more memory (cap ! nec. = size)



# vector<T>

---

- Agregar elementos
  - `v.push_back(42);` // increases size
  - `v[0] = 31;` // only ok if `v.size() >= 1`
  - `v.insert(iter before, T val)` // skipped
  - `v.insert(iter before, iter start, iter end)` // skipped
- Acceder a los elementos
  - `v.at(i)` // reference, range checked!
  - `v[i]` // reference, not range checked
  - `v.front()` // reference to first element
  - `v.back()` // reference to last element
  - `v.back() = 4;` // legal if `size > 0`



# vector<T>

---

## ■ Eliminar elementos

- `v.pop_back()`: removes last element, returns nothing
- `v.clear()`: removes everything
- `v.erase(iterator i)` // skipped
- `v.erase(iter start, iter end)` // skipped





# vector<T>

---

- Eficiencia (demora):
  - El tiempo para la inserción de elementos al principio o eliminación al final es constante.
  - Tiempo de inserción lineal para cualquier elemento del medio.
- The “standard” container



# vector<T> example

---

```
vector<char> v;  
for (int i = 0; i < 10; ++i)  
    v.push_back('A' + i);  
cout << v[0] << v.back() << endl; // AJ  
v.pop_back(); // doesn't return anything  
cout << v.size() << v.back() << endl;  
    // 9I  
for (size_t i = 0; i < v.size(); ++i)  
    cout << v[i]; // ABCDEFGHI (no J)  
cout << endl;
```



# Una referencia a algo que veremos más adelante: Iterators

- `v.begin()` y `v.end()` devuelven iterators
- Funcionan como los punteros: Incorporando funciones aritméticas (`++`, `--`) y el operador de desreferencia (`*`)

```
for (vector<char>::iterator i =  
    v.begin(); i != v.end(); ++i)  
    cout << *i; // ABCDEFGHI (no J)
```



# list<T>

---

- Bidireccional, lista lineal
- Sólo permite acceso secuencial (no L[52])
- Constructores
  - list<T>()
  - list<T>(size\_t num\_elements)
  - list<T>(size\_t num, T init)
- Métodos/Propiedades
  - l.empty() // true si l tiene 0 elementos
  - l.size() // número de elementos



# list<T>

---

- Agregado de elementos

- l.push\_back(43);
- l.push\_front(31);
- l.insert(iter b, iter s, iter s) // y otros ...

- Acceso a elementos

- l.front() // T &
- l.back() // T &
- l.begin() // list<T>::iterator
- l.end() // list<T>::iterator



# list<T>

---

## ■ Eliminación de elementos

- `l.pop_back()` // no retorna nada
- `l.pop_front()` // no retorna nada
- `l.erase(iterator i)`
- `l.erase(iter start, iter end)`

## ■ Eficiencia

- Demora prácticamente constante del tiempo de inserción y eliminación de elementos al comienzo, al final, o en el medio (debido a que pasa un iterador como argumento).



# list<T>

---

- Otros operations
  - l.sort(), l.sort(CompFn) // ordena en el mismo espacio de memoria.
  - l.splice(iter b, list<T>& grab\_from)



# list<T>

---

## ■ Ejemplo:

```
list<char> l;  
for (int i = 0; i < 4; ++i)  
{  
    l.push_front(i + 'A');  
    l.push_back(i + 'A');  
}  
for (list<char>::iterator i = l.begin();  
     i != l.end(); ++i)  
    cout << *i; // DCBAABCD
```





# Revisemos un poco!

- ¿Qué podemos hacer con todos?
  - Algunas funciones andan como (c.begin(), c.end()) son las mismas! Por lo tanto se puede escribir una función display "genérica" ...
  - Ahora vamos a mostrar un container "conocido" en 1 linea

```
template<typename Container>
void display(Container & c)
{ for(Container::iterator i = c.begin();
    i != c.end(); ++i)
    cout << *i << " ";
  cout << endl;
}
```

Dejemos esto acá para luego usarlo...



# deque<T>

---

- Colas dobles (2 inicios y dos finales)
- Acceso al azar
  - List: no es posible usar `d[i]`
  - Vector: no puen usar `push_front`
- Indicadas para agregar/remover elementos al comienzo.



# deque<T>:

## sembradores/consumidores

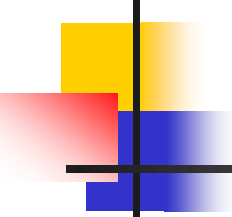
```
deque<char> d; srand(1337);  
for (int i = 0; i < N; ++i)  
{ if (rand() % 2) // agrega una letra al azar  
  al final  
    d.push_back((rand() % 26) + 'A');  
  else if (d.size()) // toma una del frente  
    d.pop_front();  
  if (d.size())  
    display(d);  
}  
cout << d.size() << " elementos restantes!" <<  
  endl;  
cout << "final: "; display(d);
```



# deque<T> example output

---

- N
- N B
- B
- B I
- B I F
- B I F F
- B I F F K
- I F F K
- I F F K H
- F F K H
- F K H
- K H
- K H X
- K H X K
- K H X K F
- H X K F
- X K F



# deque<T>: eficiencia

---

- Tiempo constante de inserción y eliminación de elementos al principio o al final de la secuencia.
- Tiempo lineal para la inserción o eliminación de elementos del medio.



# queue<T>

---

- First-in, first-out (FIFO) only
  - front(), back(), push(), pop()
  - No q[i]
  - No iterators (need to write display\_queue)
- Es un “adaptor”: que usa una deque internamente
  - Podría usar una lista (ads/disads):  
queue<T,list<T> >
- Algunas veces funciona como deque



# queue<T> example

```
queue<char> q;
srand(1337);
for (int i = 0; i < N; ++i)
{
    if (rand() % 2) // add a random letter to the back
        q.push((rand() % 26) + 'A'); // was d.push_back()
    else if (q.size()) // take the one in the front
        q.pop(); // was d.pop_front()
    if (q.size())
        display_queue(q);
}
cout << q.size() << " elements left!" << endl;
cout << "final: "; display_queue(q);
```

Same output as before!



# display\_queue

- Por qué no podemos usar `display<Container>`? No hay iterators en `queue`!
- Necesitamos una nueva funcion...
  - Q se expandirá para `queue<T>` o `queue<T, list<T> >` o cualquiera que se necesite.

```
template<typename Q>
void display_queue(const Q & q)
{ Q qc(q); // make copy
  while (!qc.empty())
  {   cout << qc.front() << " ";
      qc.pop();
  }
  cout << endl;
}
```





# stack<T>

---

- Last in, first out (LIFO)
- push(), pop(), top(), size()



# Review (so far)

---

- vector: arreglo dinámico, acceso al azar
- list: lista ligada, acceso unidireccional
- deque: queue cola doble, acceso al azar
- queue: como deque, pero sólo push, pop, front, back
- priority\_queue: queue “ordenada”, sólo push, pop, top
- stack: LIFO, push, pop, top
- Tomémosno un respiro: sólo 5 minutos 😊



# [hash\_]map, [hash\_]multimap

- Un map es un contenedor asociativo
- Dado un valor, él encontrará el otro
  - `map<string, int>` es un mapeo de strings a int's
  - maps son 1:1, multimap son 1:n
- `map`, `multimap` son logarítmicos cuando insertan/borran
  - Es necesario que se mantengan ordenados
- `hash_map`, `hash_multimap` son mejoras de tiempo constantes
  - No están ordenados (están "hasheados")



# Mmm... pairs...

---

- `pair<KeyType, ValueType>` se usan con los maps
- Se pueden crear pares `<K,V>` de objetos con `make_pair(k, v)`:

```
pair<string, int> si("foo", 42);  
si = make_pair("bar", 24);
```



# Cómo ver los pares genéricos?

```
template <typename T1, typename T2>
ostream& operator<<(ostream& o, const pair<typename T1,
    typename T2> p)
{ o << "(" << p.first << ", " << p.second << ")";
  return o;}
```

- `cout << make_pair(`  
    `make_pair("str", 31),`  
    `make_pair(Job("kernel", 1, 3), 9)`  
    `) << endl;`
- Salida: `((str, 31), (kernel user=1, pri=3, 9))`
- Además: `typeid(T1).name()` es el nombre del tipo de dato (double, int, etc.)



# Funciones de los map

---

- `m.insert(make_pair(key, value));` // inserts
- `m.count(key);` // times occurs (0, 1)
- `m.erase(key);` // removes it
- `m[key] = value;` // inserts it into the table
- `m[key]` // retrieves or creates a “default” for it
- `m.begin(), m.end()` // iterators



# Map ejemplo

```
typedef pair<string, int> dorm_location;
map<string, dorm_location> people;
people.insert(make_pair("jrl7", dorm_location("staley",
810)));
cout << people["jrl7"] << endl; // (staley, 810)
cout << people["JRL7"] << endl; // (, 0)
people["jrl7"] = dorm_location("pierce", 411);
for (map<string, dorm_location>::iterator
    p = people.begin(); p != people.end(); ++p)
    cout << "Name: " << p->first << ": " << p->second <<
endl;
// Name: JRL7: (, 0)
// Name: jrl7: (pierce, 411)
```

- Note la salida de `people["JRL7"]` insertado en el map!
- Observe cómo se sobrescribió la dirección staley por pierce.
- `p->first == (*p).first` (Ya lo sabemos de C++ ;- )



# Multimap

---

- Como map, pero con relación 1:n
- Introduce nuevas funciones:
  - `mm.upper_bound(key)` es un iterator que apunta después de todos los elementos especificados
- No permite indizado con `[ ]`, en cambio
- `mm.count(key)` retornará (0, 1, .. N)





# Multimap example

```
multimap<string, int> mm;  
mm.insert(make_pair("PSCL", 102));  
mm.insert(make_pair("EECS", 314));  
mm.insert(make_pair("EECS", 430));  
mm.insert(make_pair("EECS", 345));  
mm.insert(make_pair("PSCL", 101));  
multimap<string, int>::iterator i =  
    mm.find("EECS");  
if (i != mm.end()) //Si encontramos algún EECS  
do  
{   cout << i->second << endl;  
    ++i;  
} while (i != mm.upper_bound("EECS"));
```

Output:  
314  
430  
345



# Multimap example (continued)

```
// display all; could use display(mm) !
for (i = mm.begin(); i != mm.end(); )
{ const multimap<string, int>::iterator
  j = mm.upper_bound(i->first);
  cout << i->first << ' ' << mm.count(
    i->first) << ":";
  do
  { cout << i->second << " ";
    ++i;
  } while (i != j && i != mm.end());
  cout << endl;
}
```

Output:

```
EECS 3:314 430 345
PSCL 2:102 101
```



# Repaso

---

- Hasta ahora hemos visto 6 contenedores
- Todos ellos son muy parecidos
- Podemos recorrerlos con iteradores de manera similar



# Hash\_{...}

---

- Hay hash\_map, hash\_multimap, hash\_set, hash\_multiset
- Basicamente demoran el mismo tiempo para la inserción/eliminación en vez de tiempo log
  - No mantienen la lista ordenada
  - Son muy eficientes



# Hash performance

---

- Llene con 100,000 elementos al azar
- Busque 200,000 elementos al azar
  
- map:           llenado           0.59967s
- map:           búsqueda        1.57483s
- hash\_map:   llenado           0.615407s
- hash\_map:   búsqueda        0.872557s
- Por lo que, si no necesita el orden, trabaje con hash\_map



# Clases en contened. asociativos

- Siempre puede colocar clase en contenedores asociativos
- Deben tener un constructor por defecto que no use argumentos.
- Debe tener un operador global `bool operator<(T a, T b)`
- Puede necesitar `==` también
- No ponga punteros en ellos

```
list<string*> ss;  
ss.insert(new string("foo"));  
ss.insert(new string("foo"));  
cout << ss.size() << endl; // outputs 2!
```



# Resumen

---

- map: 1:1, ordenado,  $m[k] = v$
- multimap: 1:n, ordenado,  
`mm.insert(make_pair(k,v))`
- set: elementos únicos, ordenados
- multiset: se permiten múltiples claves,  
ordenados
- hash\_: rápido, pero no ordenado.



# Iterators

---

- Lo vimos recién
- Un iterador es como un puntero
- Se pueden incrementar para ir al “próximo” elemento.
- Algunas veces se puede avanzar o retroceder varios pasos (sumando o restando N)
- Se lo puede desreferenciar
- Existen diferentes tipos de iteradores
- Son muy útiles cuando se los combina con algorithms.





# Iterators

---

- `c.begin()` = start
- `c.end()` = 1 pasado el último elemento
  - Nunca se debe desreferenciar end! (`*c.end()` es un error!)
- Utilice mejor `++i` debido a que `i++` hace un objeto temporal y lo retorna, incrementándolo después.
- Se declara de la forma
  - `contenedor<T>::iterator i_standard`



# Diferentes tipos

---

- Tecnicamente:

- Acceso al azar ( $i += 3$ ;  $--i$ ;  $++i$ )
- Bidireccionales ( $++i$ ,  $--i$ ), store/retrieve
- Hacia adelante ( $++i$ ), store/retrieve
- entrada ( $++i$ ) retrieve
- salida ( $++o$ ) store



# iterators en la práctica

---

- iterator
  - “Standard”, va desde el principio al final
  - `c.begin()`, `c.end()`
- `const_iterator`
  - Como iterator, pero no se pueden hacer cambios (mejor!)
  - `c.begin()` and `c.end()` están sobrecargados de manera que pueden ser usados para asignar su resultados a un `const_iterator`
- `reverse_iterator`
  - Van desde el final hasta el principios con la misma semántica que los iteradores
  - Generalmente, `c.rbegin()` y `c.rend()`
  - `list`, `vector`, `deque`, `map`, `multimap`, `set`, `multiset`, `hash_`, `string`



# Iterator ejemplo

```
vector<int> v;
for (int k = 0; k < 7; ++k) v.push_back(k);
display(v); // 0 1 2 3 4 5 6
for(vector<int>::iterator i = v.begin(); i != v.end();
    ++i)
    *i = *i + 3; // Agrega 3 al contenido
display(v); // 3 4 5 6 7 8 9
for(vector<int>::const_iterator ci = v.begin(); ci !=
    v.end(); ++ci)
    cout << *ci << ' '; // *ci = *ci - 3; Qué pasa??
cout << endl; // 3 4 5 6 7 8 9
for (vector<int>::reverse_iterator ri = v.rbegin(); ri
    != v.rend(); ++ri)
{
    *ri = *ri - 3;
    cout << *ri << ' ';
}
cout << endl; // 6 5 4 3 2 1 0
```



# Algorithms

---

- AJA: al final, la mayor parte 😊
- Basicamente, la STL provee 70 algorithms de interés general
- Muchos de ellos toman iteradores y retornan iteradores



# Categorías de algorithms

---

- Copia
- Búsqueda en secuencias desordenadas
- Búsqueda en secuencias ordenadas
- Reemplazo/eliminación de elementos
- Reordenamiento
- Ordenamiento
- Mezcla secuencias ordenadas
- Operaciones con conjuntos
- Operaciones con montículos
- Mínimo y máximo
- Permutaciones
- Transformar y generar
- Numéricos
- Otros



# All algorithms

---

- adjacent\_find, binary\_search, copy, copy\_backward, count, count\_if, equal, equal\_range, fill, fill\_n, find, find\_end, find\_first\_of, find\_if, for\_each, generate, generate\_n, includes, inplace\_merge, iter\_swap, lexicographical\_compare, lower\_bound, make\_heap, max, max\_element, merge, min, min\_element, mismatch, next\_permutation, nth\_element, partial\_sort, partial\_sort\_copy, partition, pop\_heap, prev\_permutation, push\_heap, random\_shuffle, remove, remove\_copy, remove\_copy\_if, remove\_if, replace, replace\_copy, replace\_copy\_if, replace\_if, reverse, reverse\_copy, rotate, rotate\_copy, search, search\_n, set\_difference, set\_intersection, set\_symmetric\_difference, set\_union, sort, sort\_heap, stable\_partition, stable\_sort, swap, swap\_ranges, transform, unique, unique\_copy, upper\_bound
- accumulate, adjacent\_difference, inner\_product, partial\_sum



# Algorithms más famosos

---

- copy, find, remove, unique, reverse, sort, min/max\_element, count, accumulate, for\_each, transform, equal
- Puede escribir los suyos propios...





# copy(b, e, Dest)

---

```
vector<int> w, v;  
w.push_back(42); w.push_back(31);  
  w.push_back(1);  
v.resize(w.size()); // needed  
copy(w.begin(), w.end(), v.begin());  
vector<int> c;  
copy(w.begin(), w.end(), back_inserter(c));  
  // calls c.push_back  
deque<int> d;  
copy(w.begin(), w.end(), front_inserter(d));  
  // calls d.push_front  
display(v); // 42 31 1  
display(c); // 42 31 1  
display(d); // 1 31 42 (b/c added to front)
```



# copy + istream iterators

---

- Un iterador istream lee desde un istream (file, console, etc.) hasta llegar a un eof
- Podemos hacer cosas como ésta COQUETO!:

```
vector<int> v;  
copy(istream_iterator<int>(ifstream("numbers.txt")),  
      istream_iterator<int>(), back_inserter(v));
```

- Note que el iterador “end”  
`istream_iterator<int>()` ; una instancia especial que indica el EOF



# copy + ostream iterators

- Veamos ahora los ostream\_iterators:
- `copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));`
- Ahora podemos reescribir display en 2 líneas, sin ningún bucle:

```
template<typename Container>
void display(const Container & c, string sep = " ")
{ copy(c.begin(), c.end(), ostream_iterator<typename
  Container::value_type>(cout, sep.c_str()));
  cout << endl;
}
```



# i = find(b, e, val)

---

- Retorna un iterador (o end si no lo encuentra)

```
if (find(v.begin(), v.end(), 31)
    != v.end())
    cout << "Found 31." << endl;
```



# ne = remove(b, e, val)

- No puede modificar en el momento el puntero al final: retorna el nuevo final.
- `remove_copy(b, e, Dest, val)` removerá el valor copiando el resultado en Dest

```
v.clear();  
for(int i = 0; i < 10; ++i)          v.push_back(i);  
vector<int> vc;  
remove_copy(v.begin(), v.end(), back_inserter(vc), 4);  
display(vc); // 0 1 2 3 5 6 7 8 9  
  
vector<int>::const_iterator e = remove(v.begin(), v.end(), 4);  
display(v); // 0 1 2 3 5 6 7 8 9 9 "extra 9"  
  
for (vector<int>::const_iterator vi = v.begin();  
     vi != e; ++vi) // not v.end()  
    cout << *vi << ' ';  
cout << endl; // 0 1 2 3 5 6 7 8 9
```



# sort(b, e)

---

- Ordena un rango; requiere iteradores de acceso al azar
- $N \log N$ , promedio y peor caso
- `sort(v.begin(), v.end());`
- Se generará un error al momento de la compilación si no estuviera soportado
- Podemos ordenar arreglos C de esta manera también

```
int A[] = {1, 4, 2, 8, 5, 7};
```

```
const int N = sizeof(A) / sizeof(int);
```

```
sort(A, A + N);
```



# reverse(b, e)

---

- Bidirectional iterators
- Reverses the range in-place
- Linear

```
for(int i = 0; i < 10; ++i)  
    v.push_back(i);
```

```
display(v); // 0 1 2 3 4 5 6 7 8 9
```

```
reverse(v.begin(), v.end());
```

```
display(v); // 9 8 7 6 5 4 3 2 1 0
```



# min/max\_element(b,e)

- An iterator to the minimum/maximum element in a range (e if range is empty) (\*min\_element(..) is value)
- Doesn't need to be sorted

```
list<int> l;
```

```
l.push_back(4); l.push_front(12);
```

```
l.push_back(1); l.push_front(2);
```

```
l.push_back(7);
```

```
display(l); // 2 12 4 1 7
```

```
cout << *min_element(l.begin(), l.end()) <<  
endl; // 1
```

```
cout << *max_element(l.begin(), l.end()) <<  
endl; // 12
```





# count(b, e, V)

---

- Cuenta las ocurrencias de V en el rango [b,e]
- count\_if(b, e, Func):

```
bool iscomma(char c)
```

```
{
```

```
    return c == ',';
```

```
}
```

```
string s = "(X,Y) or [X,Y]?";
```

```
cout << count_if(s.begin(), s.end(),  
    iscomma) << endl;
```

Outputs 2



# accumulate(b, e, initial)

- in <numeric>
- Adds (+) up the values in the range b, e
  - Initial is used if it's an empty range
  - As long as operator+ is defined, we're good
  - You can pass in a binary function to use instead of + (beyond scope)

```
vector<string> vs;  
vs.push_back("This"); vs.push_back("Talk");  
vs.push_back("Is"); vs.push_back("Way");  
vs.push_back("Too"); vs.push_back("Long");  
cout << accumulate(vs.begin(), vs.end(),  
    string("")) << endl; // ThisTalkIsWayTooLong
```



## for\_each(b, e, op)

---

- Puede `op(*i)` para cada `i` en el rango `b, e`;  
MAGICO @@

```
void display_backwards(string s)
{ // strings have reverse iterators
  copy(s.rbegin(), s.rend(),
        ostream_iterator<char>(cout));
}

for_each(vs.begin(), vs.end(),
        display_backwards);

// sihTklaTsIyaWooTgnol
```



## for\_each(b, e, op)

---

- Using “function objects”, you can do some neat stuff

```
void truncate_word(string s, int  
    len)
```

```
{ cout << s.substr(0, len) ;  
}
```

```
for_each(vs.begin(), vs.end(),  
    bind2nd(ptr_fun(truncate_word),  
    2)) ;
```

```
// ThTaIsWaToLo
```



# transform

- La favorita; probablemente el algoritmo más potente.
- Dos formas: `transform(b, e, out, UnaryFunc)`:
  - Aplica `UnaryFunc` a `b`, `e` y almacena el resultado en `out`

```
string s = "hello";
string upcase, upcasebackwards;
transform(s.begin(), s.end(), back_inserter(upcase),
          toupper);
// reverse and upper case in one line
transform(s.rbegin(), s.rend(),
          back_inserter(upcasebackwards), toupper);
cout << upcase << " " << upcasebackwards << endl;
// HELLO OLLEH
// straight to screen
transform(s.rbegin(), s.rend(),
          ostream_iterator<char>(cout), toupper); // OLLEH
cout << endl;
```



# transform

- transform(Argument 1 beginning, Arg 1 end, Argument 2 beginning, output, BinaryFunction);

```
typedef pair<double, double> point;
point midpoint(point p1, point p2)
{ return point((p1.first + p2.first)/2.0,
               (p1.second + p2.second)/2.0);
}
vector<point> v1, v2;
v1.push_back(point(0.0, 3.1));
v1.push_back(point(7.3, 8.1));
v1.push_back(point(9.1, -2.89));
v2.push_back(point(4.3, 9.3));
v2.push_back(point(1.3, 9.1));
v2.push_back(point(8.3, -3.1));
display(v1); // (0, 3.1) (7.3, 8.1) (9.1, -2.89)
display(v2); // (4.3, 9.3) (1.3, 9.1) (8.3, -3.1)
transform(v1.begin(), v1.end(), v2.begin(),
          ostream_iterator<point>(cout, " "), midpoint);
// (2.15, 6.2) (4.3, 8.6) (8.7, -2.995)
```



# Revisión

---

- Revisemos los ejemplos del inicio de la presentación
- Problema 1:
  - Leer una lista de números y obtener valores mínimo/máximo, promedio, media geométrica y mediana.
- Problema 2:
  - Leer una lista de palabras y como salida mostrar la cantidad de veces que cada una de ellas aparece



# Example 1: STL solution

---

```
vector<int> v;
```

```
copy(istream_iterator<int>(ifstream("numbers.txt")),  
     istream_iterator<int>(), back_inserter(v));
```

```
sort(v.begin(), v.end());
```

```
cout << "min/max: " << v.front() << " " << v.back() << endl;  
// could have used min/max_element but list already sorted  
cout << "median : " << *(v.begin() + (v.size()/2)) << endl;
```

```
cout << "average: " << accumulate(v.begin(), v.end(), 0.0)  
    / v.size() << endl;
```

```
cout << "geomean: " <<  
    pow(accumulate(v.begin(), v.end(), 1.0, multiplies<double>()),  
        1.0/v.size()) << endl;
```





# Example 2: STL solution

---

```
vector<string> v;
```

```
map<string, int> m;
```

```
copy(istream_iterator<string>(ifstream("words.txt")),  
      istream_iterator<string>(), back_inserter(v));
```

```
for (vector<string>::iterator vi = v.begin();  
     vi != v.end(); ++vi)  
    ++m[*vi];
```

```
for (map<string, int>::iterator mi = m.begin();  
     mi != m.end(); ++mi)  
    cout << mi->first << ": " << mi->second << endl;
```



# Conclusion

---

- La STL tiene de todo
- Haga que el compilador trabaje por usted
- Ahorre tiempo y líneas de código
- Próximos pasos:
  - Revise los apuntes de STL
  - Use STL en sus trabajos