

<div data-bbox="61 304 466 346" data-label="Section-Header"><h1>Standard Template Library</h1></div> <div data-bbox="61 441 466 571" data-label="Text"><p>Di Paola Martín martinp.dipaola &lt;at&gt; gmail.com</p><p>Facultad de Ingeniería Universidad de Buenos Aires</p></div> <div data-bbox="779 699 792 720" data-label="Page-Footer"><p>1</p></div>	<div data-bbox="834 153 1039 186" data-label="Section-Header"><h2>De qué va esto?</h2></div> <div data-bbox="875 350 1161 520" data-label="Text"><p>Standard Template Library</p><p>Containers</p><p>Iteradores</p><p>Algoritmos</p></div> <div data-bbox="1591 699 1604 720" data-label="Page-Footer"><p>2</p></div>
<div data-bbox="136 947 540 1045" data-label="Section-Header"><h1>Standard Template Library</h1><h2>Containers</h2></div> <div data-bbox="779 1308 792 1329" data-label="Page-Footer"><p>3</p></div>	<div data-bbox="834 762 1515 795" data-label="Section-Header"><h2>Containers - Programar en C++ y no en C con objetos</h2></div> <div data-bbox="875 875 1477 936" data-label="Text"><p>C es muy eficiente, pero tareas simples pueden resultar titánicas.</p></div> <div data-bbox="899 963 1373 1117" data-label="List-Group"><ul style="list-style-type: none"><li>• Manejo de texto.</li><li>• Armar un vector que aumente de tamaño.</li><li>• Frecuencia de elementos.</li><li>• Remover duplicados.</li></ul></div> <div data-bbox="875 1176 1555 1239" data-label="Text"><p>C++ ofrece containers muy versátiles y eficientes. En C++, hay que programar en C++ y <b>no en C!</b></p></div> <div data-bbox="1591 1308 1604 1329" data-label="Page-Footer"><p>4</p></div>
<div data-bbox="24 1371 433 1404" data-label="Section-Header"><h2>Manejo de textos con std::string</h2></div> <div data-bbox="61 1476 727 1539" data-label="Text"><p>Útil para el manejo de textos pero no para el manejo de blobs binarios.</p></div> <div data-bbox="24 1560 578 1835" data-label="Code-Block"><pre>1 std::string saludos = "Hola_mundo!"; 2 3 //Substring "ola" 4 std::string otro_string = saludos.substr(1, 3); 5 6 // Comparacion de strings 7 bool son_iguales = (saludos == otro_string); 8 9 // Concatenacion 10 otro_string = "H" + otro_string + "_mundo!";</pre></div> <div data-bbox="779 1917 792 1938" data-label="Page-Footer"><p>5</p></div>	<div data-bbox="899 1398 1550 1543" data-label="List-Group"><ul style="list-style-type: none"><li>• <code>std::string</code> es un container flexible y poderoso pensado en el procesamiento de texto.</li><li>• Una alternativa es <code>std::vector&lt;char&gt;</code>. Ambos containers tienen sus pro y contras. Ver que métodos soportan cada uno y tomar la decisión en función de ello.</li></ul></div>

Adios los new[] con std::vector

Por ser RAII, es una alternativa al `new[]` (excepto para `Vector<bool>`)  
`std::vector<char>` es una muy buena elección para manejar blobs binarios pero no para manejo de textos.

```
1 std::vector<char> data(256, 0);
2
3 char *buf = data.data();
4 file.read(buf, 256);
```

6

- El método `vector::data` nos da acceso al almacenamiento interno del vector y nos permite interactuar con la api de C que espera un buffer de tipo `char*`.
- Por qué excepto para `vector<bool>`? por que esa clase es una especialización total como la que vimos aquí.

Cálculo de frecuencias con std::map (Arrays asociativos)

```
1 std::map<char, int> freq_de_caracteres;
2
3 std::string texto = "Lorem ipsum dolor sit amet,_" /*...*/
4
5 for (int i = 0; i < texto.length(); ++i) {
6     char c = texto[i];
7     if (freq_de_caracteres.count(c)) {
8         freq_de_caracteres[c] += 1;
9     }
10    else {
11        freq_de_caracteres[c] = 1;
12    }
13 }
14
15 // vease tambien su version hash
16 std::unordered_map<K, V>
```

7

Remove duplicados con std::set

```
1 void remover_duplicados(std::list<int> &lista) {
2     std::set<int> unicos(lista.begin(), lista.end());
3     std::list<int> filtrado(unicos.begin(), unicos.end());
4
5     lista.swap(filtrado);
6 }
7
8 // vease tambien la version hash de set
9 std::unordered_set<K, V>
```

8

- Muchos containers pueden construirse a partir de otros a traves de dos iteradores que marcan desde donde y hasta donde se deben copiar los elementos.
- Dado que `std::set` ignora los duplicados esto es una forma interesante de resolver el problema.
- Nota: como side effect, `std::set` nos dejara ordenados los elementos que no fueron removidos.
- Para finalizar se podría haber hecho `lista = filtrado`; pero eso generaría otra copia mas. El método `swap` cambia los containers internos y resulta mas eficiente que una copia.

Y los clásicos de hoy y de siempre

```
1 std::list<int> lista; // doubled "linked" list
2
3 lista.push_back(1); lista.push_front(2);
4 lista.insert(...); lista.erase(...);
5
6 std::stack<int> pila;
7
8 pila.push(1);
9 pila.pop(); // no devuelve nada!
10
11 std::queue<int> cola;
12
13 cola.push(1);
14 cola.pop(); // pull (no devuelve nada!)
15
16 // Para obtener el valor de un stack/queue
17 int i = pila.top(); int j = cola.front();
```

9

- Los clásicos, lista, colas, pilas (incluso hay colas prioritarias)
- `back()`, `front()` y `top()` retornan una referencia del primer y último elemento mientras que `pop()` y `push()` los remueven pero sin devolverlos. El por qué de esta interfaz que separa el retorno de la remoción es debido a las garantías Exception Safety (que las veremos en la clase de Excepciones y Manejo de Errores en C++).

## Custom: Containers, adapters y allocators

```
1 std::stack<int> pila;
2
3 std::stack<int, std::vector<int>>> pila;
4
5 std::stack<int, std::vector<int, MyAlloc<int>>>> pila;
```

- Algunos containers son en realidad adapters y podemos cambiar el container real que usan detras de escena.
- Más aun, podemos cambiar en donde allocan los objetos los containers: no usan `new` directamente sino que usan un alocador, un objeto que podemos cambiar.
- Como la customización se hace a traves de un parámetro template esta se hace en tiempo de compilación y no conlleva ningun overhead en runtime.

10

- Algunos containers son en realidad adapters. Esto significa que podemos cambiar el objeto interno que implementa realmente el container.
- Aun mas, los containers usan un objeto "allocator" por default para acceder a la memoria (no hacen `new` sino que llaman a un método `allocate`). Este objeto allocator se puede cambiar tambien! Se podría implementar un stack que en vez de usar la memoria use un archivo o una shared memory.

## STL- Resumen

- Esto no es C. Hay una lib estándar más completa. Usarla. Un buen sitio para buscar info es <http://www.cplusplus.com/reference/>
- Los containers pueden ser muy eficientes si los eligen correctamente.

11

## Standard Template Library

### Iteradores

## Iteradores - Abstracción del container

Muchos algoritmos son independientes del container sobre el que trabajan; sólo necesitan una forma de recorrerlos.

- Sumatoria de números de un container.
- Imprimir sus elementos.
- Búsqueda secuencial.

Los iteradores abstraen la forma de recorrer un container. En C++, hay distintas clases de iteradores pero cada container sólo implementa aquellos que pueda hacerlos **eficientemente**.

12

13

## Iteradores

```
1 // Todos pueden
2 ++it;      it++;
3 it = itx;   Iter it(itx);

4 // Input (mutables)      4 // Output (inmutables)
5 *it = t;    *it++ = t;    5 t = *it;    it->m;
6 //          6 it == itx; it != itx;

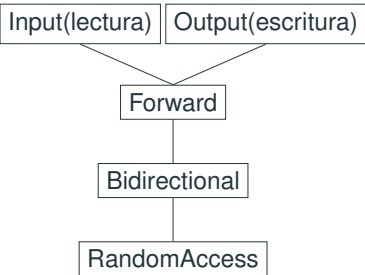
7 // Bidirectional
8 --it;      it--;

9 // RandomAccess (aka pointers)
10 it + n;    it - n;    it[n];
11 it += n;   it + itx;  it < n;
```

14

- Todos los iteradores pueden moverse hacia adelante (forward) y copiarse.
- Algunos soportan el dereferenciado para la asignación (input) mientras que otros lo soportan para la lectura del objeto apuntado (output). Estos últimos permiten realizar comparaciones por igualdad.
- Otros soportan moverse hacia atrás (backward)
- Y finalmente existen algunos iteradores que permiten moverse de forma no secuencial como si fueran punteros.

## Iteradores



15

- Existe una jerarquía de iteradores. Todos los iteradores con RandomAccess son iteradores Bidireccionales pero no todos los Bidireccionales tienen RandomAccess

## Lifetime de los iteradores

Mal, si el container es modificado los iteradores son inválidos:

```
1 std::list<int>::iterator it = lista.begin();
2 for (; it != lista.end(); ++it)
3     if (*it % 2 == 0) // remover si es par
4         lista.erase(it); // el container fue modificado!!
```

Bien:

```
1 std::list<int> tmp;
2 std::list<int>::iterator it = lista.begin();
3 for (; it != lista.end(); ++it)
4     if (*it % 2 != 0) // copiar si no es par
5         tmp.push_back(*it);
6 lista.swap(tmp);
```

Mucho mejor!:

```
1 bool es_par(const int &i) { return i % 2 == 0; }
2 std::remove_if(lista.begin(), lista.end(), es_par);
```

16

- En general un iterador es válido mientras que su container no cambie: iterar un container para removerle algunos elementos suele ser un bug clásico.

## Standard Template Library

### Algoritmos

17

## Algoritmos genéricos - Abstracción de código

```
1 // no compila por un mini detalle (typename)
2 template <class Container, class Val>
3 Container::iterator find(
4     Container &v,
5     const Val &val) {
6     Container::iterator it = v.begin();
7     Container::iterator end = v.end();
8
9     while (it != end and val != *it) {
10         ++it;
11     }
12
13     return it;
14 }
```

18

## Intermezzo: typename

```
1 struct List {
2     struct iterator {
3         /* ... */
4     };
5
6     List::iterator begin() {
7         return List::iterator(/*...*/);
8     }
9 };
```

- `List::iterator` hace referencia a un tipo (el `struct` `iterator` dentro de `List`)
- `List::begin` hace referencia a un método de `List`

19

## Intermezzo: typename

Cómo sabe el compilador que `Container::iterator` es un tipo y no un método si ni siquiera sabe que es `Container`?

```
1 template <class Container, class Val>
2 Container::iterator find(...) { ... }
```

La keyword `typename` permite diferenciar un método de un tipo.

```
1 template <class Container, class Val>
2 typename Container::iterator find(...) { ... }
```

20

## Algoritmos genéricos - Abstracción de código

```
1 // ahora si compila (siempre que Container y Val cumplan)
2 template <class Container, class Val>
3 typename Container::iterator find(
4     Container &v,
5     const Val &val) {
6     typename Container::iterator it = v.begin();
7     typename Container::iterator end = v.end();
8
9     while (it != end and val != *it) {
10         ++it;
11     }
12
13     return it;
14 }
```

21

## Algoritmos con iteradores, no con containers

```
1 template <class Iterator, class Val>
2 Iterator find(Iterator &it, Iterator &end, const Val &val) {
3     while (it != end and val != *it) {
4         ++it;
5     }
6
7     return it;
8 }
```

22

- La mayoría de los algoritmos deberían escribirse en términos de iteradores: recibir y retornar iteradores, independizándose del container en cuestión.

## Algoritmos de la STL: menos código, menos bugs!

For each, (también conocido como map)

```
1 | std::for_each(container.begin(), container.end(), func);
```

23

## Algoritmos de la STL: menos código, menos bugs!

Como imprimir al stdout un container (útil para debug)

```
1 | template<class T>
2 | void print_to_cout(const T &val) {
3 |     std::cout << val << " ";
4 | }
5 |
6 | std::list<int> l;
7 | for_each(l.begin(), l.end(), print_to_cout<int>);
```

24

## Algoritmos de la STL: menos código, menos bugs!

O con functors:

```
1 | template<class T>
2 | struct Printer {
3 |     std::ostream &out;
4 |
5 |     Printer(std::ostream &out) : out(out) {}
6 |
7 |     void operator() (const T &val) {
8 |         out << val << " ";
9 |     }
10 | };
11 |
12 | std::list<int> l;
13 | for_each(l.begin(), l.end(), Printer<int>(std::cout));
```

25

## Algoritmos de la STL: menos código, menos bugs!

Sorting

```
1 | // usando el operador less < como ordenador
2 | std::sort(container.begin(), container.end());
3 |
4 | // usando la funcion/functor especifica
5 | std::sort(container.begin(), container.end(), less_func);
6 |
7 | // orden estable
8 | std::stable_sort(container.begin(), container.end());
```

Searching (sobre containers ordenados)

```
1 | // usando la misma funcion/functor que se uso para el
2 | // ordenamiento (el operador less < es el default)
3 | std::binary_search(container.begin(), container.end(),
4 |                     val_to_be_found);
```

26

## Algoritmos de la STL: código con optimizaciones

Swap, con implementaciones especializadas para containers

```
1 | int a = 1, b = 2;
2 | std::swap(a, b);    // a == 2, b == 1 haciendo copias
3 |
4 | std::list<int> a;
5 | std::list<int> b;
6 | std::swap(a, b);    // solo swap de punteros internos!
```

27

## STL - Resumen




- El uso de templates, containers e iteradores puede dejar el código muy verbose, usar `typedef` y `using`
- Usar el operador de preincremento `++it` y no el de pos incremento para evitar copias.
- Busquen! `std::stack`, `std::queue`, `std::make_heap`, `std::set_intersection`, `std::set_union`, etc. Hay más contenedores y algoritmos listos para ser usados.  
**Encuentrenlos y usenlos!**

28

## Appendix

### Referencias

## Referencias I

-  <http://cplusplus.com>
-  Herb Sutter.  
***Exceptional C++: 47 Engineering Puzzles.***  
Addison Wesley, 1999.
-  Bjarne Stroustrup.  
***The C++ Programming Language.***  
Addison Wesley, Fourth Edition.