STL: Containers

Instituto Balseiro

STL – Containers

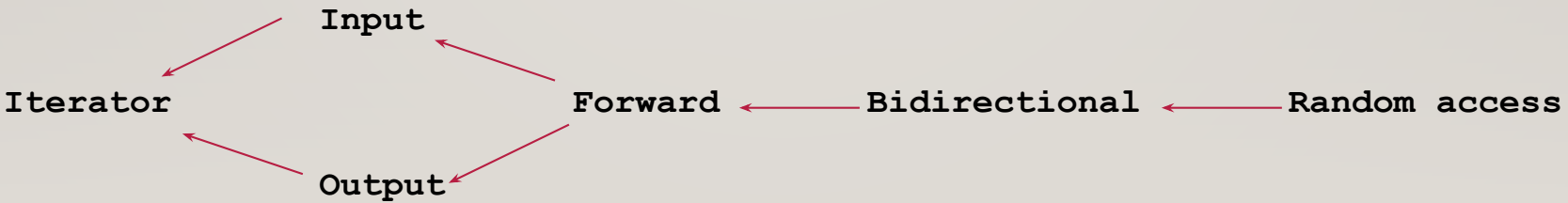
- Los contenedores son objetos que contienen otros objetos, como listas, vectores y arrays asociativos.
- Las operaciones en general son agregar, acceder y remover objetos.
- Los distintos contenedores proveen una interface común.
- La implementación de cada uno está optimizada.
- Cada contenedor provee operaciones ideales para su uso.
- Un uso común de los contenedores es iterar sobre sus elementos, para eso cada contenedor define una clase `iterator` apropiada. El usuario en general no necesita saber si los datos están en una lista o un vector.

STL – Containers iterators

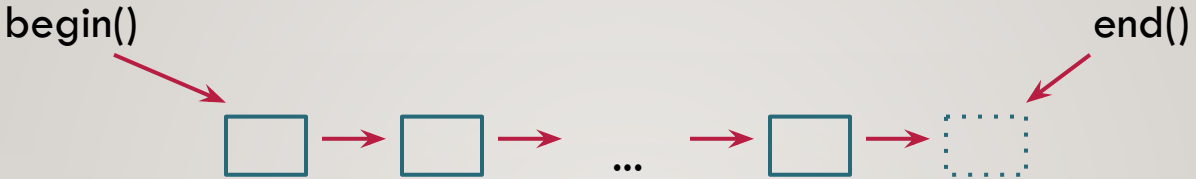
- Los iteradores dan una noción de secuencia, implementando una operación de get-next-element.
- El modelo container-iterator permite:
  - Contenedores simples y eficientes.
  - Independencia de contenedores.
  - Interface común de uso a través de iteradores.
  - Se puede definir distintos iteradores para un contenedor para diferentes necesidades.
  - La jerarquía de clases de iteradores es complicada.
  - No hay nada en común entre contenedores y objetos contenidos.

STL – Iterator Operations

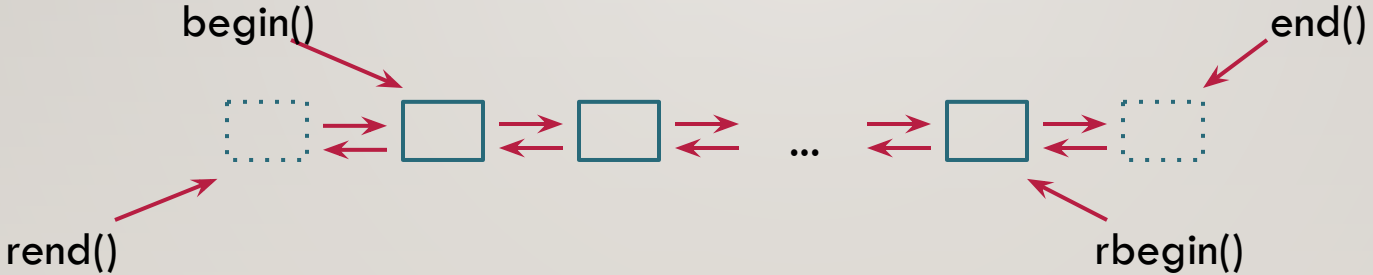
Categoría:	output	input	forward	bidirectional	random-access
Abreviación:	Out	In	For	Bi	Ran
Read:		=*p	=*p	=*p	=*p
Access:		->	->	->	-> []
Write:	*p=		*p=	*p=	*p=
Iteration:	++	++	++	++ --	++ -- + - += -=
Comparación:		== !=	== !=	== !=	== != < > >= <=



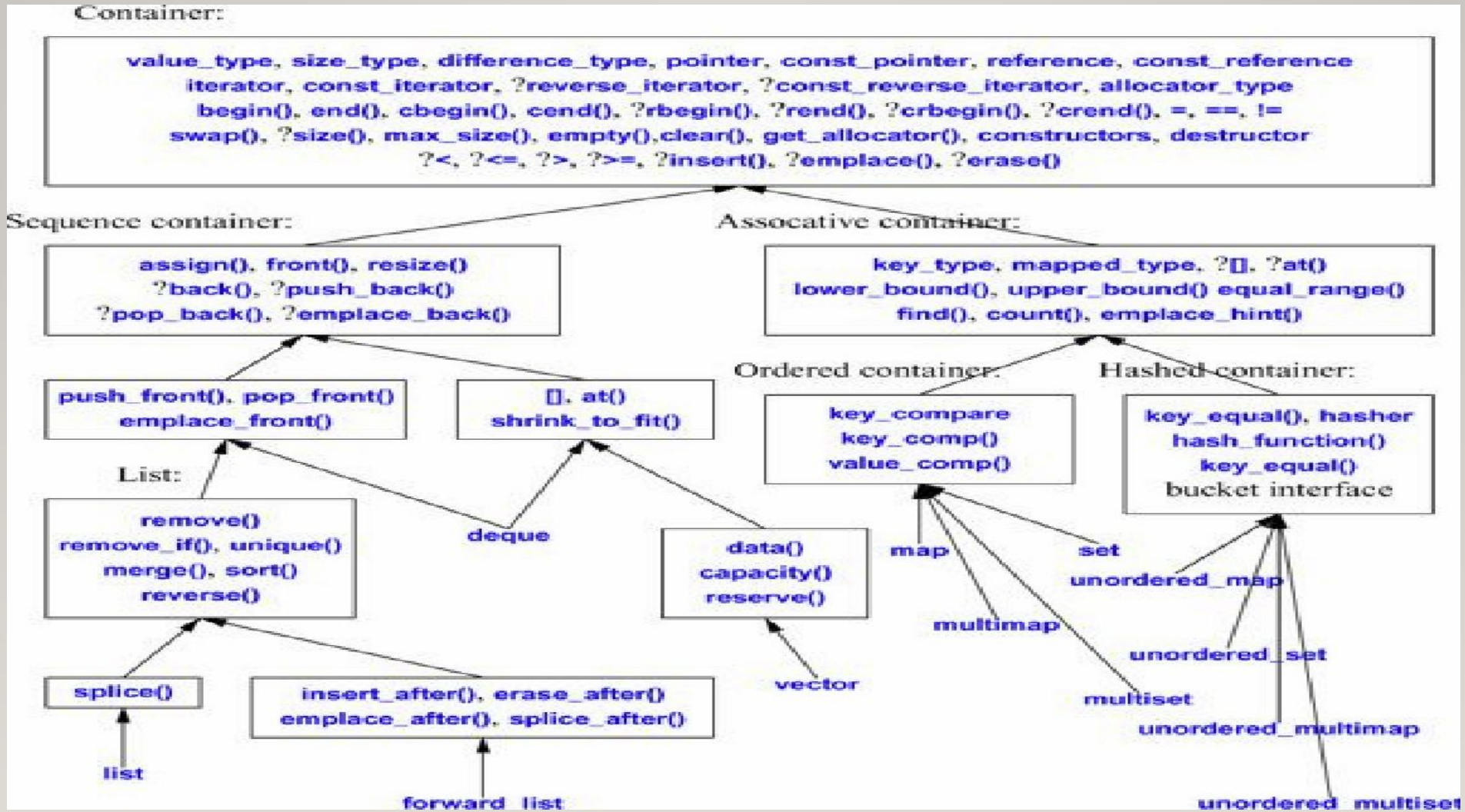
STL – Iterators



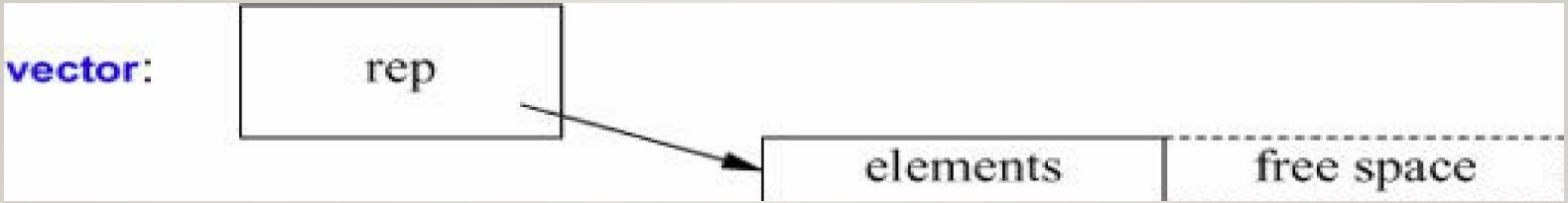
- Forward iterator

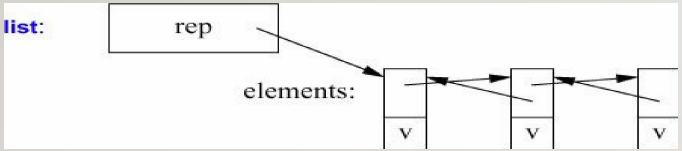


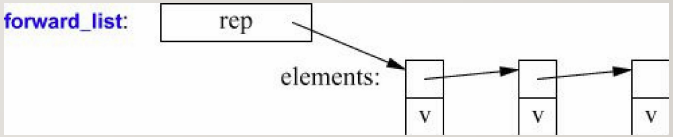
- Bidirectional iterator, reverse iterator

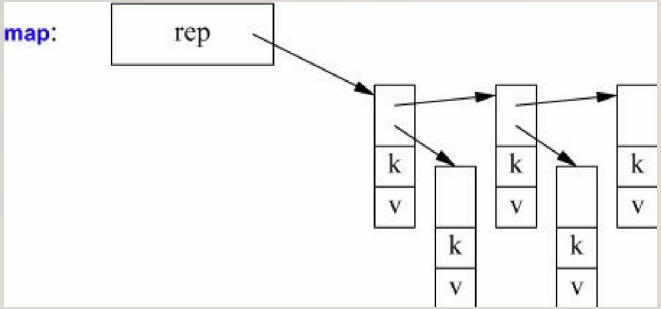

Container:
value_type, size_type, difference_type, pointer, const_pointer, reference, const_reference
iterator, const_iterator, ?reverse_iterator, ?const_reverse_iterator, allocator_type
begin(), end(), cbegin(), cend(), ?rbegin(), ?rend(), ?crbegin(), ?crend(), =, ==, !=
swap(), ?size(), max_size(), empty(),clear(), get_allocator(), constructors, destructor
?<, ?<=, ?>, ?>=, ?insert(), ?emplace(), ?erase()
Sequence container:
Assocative container:
assign(), front(), resize()
?back(), ?push_back()
?pop_back(), ?emplace_back()
key_type, mapped_type, ?[], ?at()
lower_bound(), upper_bound() equal_range()
find(), count(), emplace_hint()
push_front(), pop_front()
emplace_front()
[], at()
shrink_to_fit()
Ordered container:
Hashed container:
key_compare
key_comp()
value_comp()
key_equal(), hasher
hash_function()
key_equal()
bucket interface
List:
remove()
remove_if(), unique()
merge(), sort()
reverse()
deque
data()
capacity()
reserve()
map
set
unordered_map
multimap
unordered_set
multiset
unordered_multimap
splice()
insert_after(), erase_after()
emplace_after(), splice_after()
vector
list
forward_list
unordered_multiset

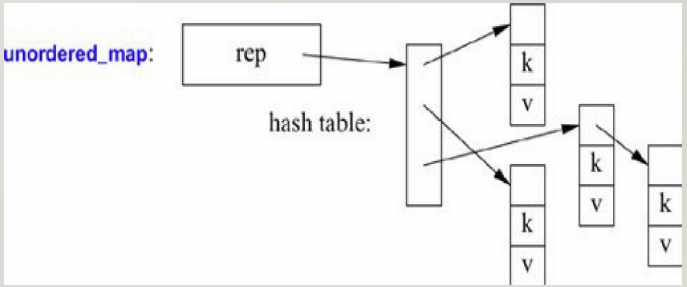
STL – Containers Representation











STL – Containers operations

Member Types	
`value_type`	Tipo del elemento
`allocator_type`	Tipo del allocator
`size_type`	Tipo de subindices, cantidad, etc. (unsigned)
`difference_type`	Tipo de la diferencia entre iteradores
`iterator`	Tipo del iterador, se comporta como `value_type *`
`const_iterator`	Tipo del iterador de un contenedor `const`
`reverse_iterator`	Tipo del iterador en orden inverso
`const_reverse_iterator`	Tipo del iterador en orden inverso, constante
`reference`	Tipo de referencia del elemento, idem `value_type &`
`const_reference`	Tipo de referencia constante del elemento
`key_type`	Tipo de clave (para containers asociativos)
`mapped_type`	Tipo de objeto mapeado (para containers asociativos)
`key_compare`	Tipo de criterio de comparación (para containers asociativos)

STL – Containers operations

Iterators	
`begin()`	Apunta al primer elemento
`end()`	Apunta al siguiente al último elemento
`rbegin()`	Apunta al primer elemento en secuencia inversa
`rend()`	Apunta al siguiente al último elemento en secuencia inversa

Element Access	
`front()`	Primer elemento
`back()`	Último elemento
`[]`	Subindexado, unchecked (no para lista)
`at()`	Subindexado, checked (no para lista)

STL – Containers operations

Stack and Queue Operations	
`push_back()`	Agrega al final
`pop_back()`	Remueve el último elemento
`push_front()`	Agrega un nuevo primer elemento (sólo para `list` y `deque`)
`pop_front()`	Remueve primer elemento (sólo para `list` y `deque`)

List Operations	
`insert(p,x)`	Agrega `x` antes de `p`
`insert(p,n,x)`	Agrega `n` copias de `x` antes de `p`
`insert(p,first,last)`	Agrega elementos de [`first`:`last`[antes de `p`
`erase(p)`	Remueve el elemento apuntado por `p`
`erase(first,last)`	Remueve [`first`:`last`[
`clear()`	Remueve todos los elementos

STL – Containers operations

Other Operations	
`size()`	Número de elementos
`empty()`	Está vacío el contenedor?
`max_size()`	Cantidad de elementos del máximo contenedor posible
`capacity()`	Espacio allocado por vector (sólo para **`vector`**)
`reserve()`	Reserva espacio para futura expansión (sólo para vector)
`resize()`	Cambia el tamaño (sólo para vector, **`list`** y **`deque`**)
`swap()`	Intercambia elementos de dos contenedor
`get_allocator()`	Retorna una copia del allocador del contenedor
`==`	Es el contenido de dos contenedores el mismo?
`!=`	Es el contenido de dos contenedores distinto?
`<`	Está un contenedor lexicográficamente antes que otro?

STL – Containers operations

Constructors	
`container()`	Contenedor vacío
`container(n)`	**n** elementos con valor default (no para cont. asociativos)
`container(n,x)`	**n** copias de **x** (no para contenedores asociativos)
`container(first,last)`	Elementos iniciales de [**first**:**last**[
`container(c)`	Constructor copia, elementos iniciales de **c**
`~container()`	Destruye el contenedor y todos sus elementos

Assignments	
`operator=(c)`	Copia los elementos de **c**
`assign(n,x)`	Asigna **n** copias de **x** (no para contenedores asociativos)
`assign(first,last)`	Asigna de [**first**:**last**[

STL – Containers operations

Associative Operations	
`operator[](k)`	Accede al elemento con clave **k** (para cont. con clave única)
`find(k)`	Busca el elemento con clave **k**
`lower_bound(k)`	Busca el primer elemento con clave **k**
`upper_bound(k)`	Busca el primer elemento con clave mayor a **k**
`equal_range(k)`	Retorna `lower_bound(k)` y `upper_bound(k)`
`key_comp()`	Copia del objeto de comparación de claves
`value_comp()`	Copia del objeto de comparación de valores mapeados

STL – Containers Complexity

Standard Container Operation Complexity					
	[] §31.2.2	List §31.3.7	Front §31.4.2	Back §31.3.6	Iterators §33.1.2
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
forward_list		const	const		For
deque	const	O(n)	const	const	Ran
stack				const	
queue			const	const	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
unordered_map	const+	const+			For
unordered_multimap		const+			For
unordered_set		const+			For
unordered_multiset		const+			For
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
built-in array	const				Ran
valarray	const				Ran
bitset	const				

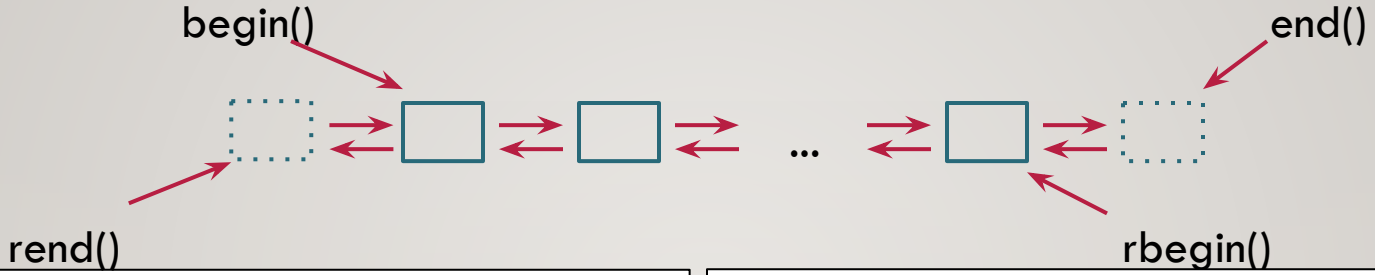
STL – Vector

```
template <class T, class A = allocator<T>> class std::vector {
    public:
        // types:
        typedef T value_type;
        typedef A allocator_type;
        typedef implementation_dependent1  size_type;
        typedef implementation_dependent2  difference_type;

        typedef implementation_dependent3 iterator;
        typedef implementation_dependent4 const_iterator;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_revese_iterator;

        typedef typename A::pointer pointer;
        typedef typename A::const_pointer const_pointer;
        typedef typename A::reference reference;
        typedef typename A::const_reference const_reference;

        // ...
};
```

STL – Vector

```
template <class T, class A = allocator<T>> class std::vector {
    public:
        // …

        // iterators:
        iterator begin();               // points to first element
        const_iterator begin() const;
        iterator end();                 // points to one-past-last element
        const_iterator end() const;

        reverse_iterator rbegin();      // points to first in reverse sequence
        const_reverse_iterator rbegin() const;
        reverse_iterator rend();        // points to one-past-last in rev. seq.
        const_reverse_iterator rend() const;

        // ...

};
```

STL – Vector iterators




```
template <class C>
typename C::iterator find_last(
              C &c,
              typename C::value_type v) {
    typename C::iterator p = c.end();
    while( p != c.begin() ) {
        --p;
        if( *p == v ) return p;
    }
    return p;
}
```

```
template <class C>
typename C::iterator find_last2(
    C& c,
    typename C::value_type v)
{
    typename C::reverse_iterator p =
                        c.rbegin();
    while (p != c.rend()) {
        if (*p == v)
            return prev(p.base());
        p++;
    }
    return c.begin();
}
```

```
int main()
{
    list<int> v{ 1, 2, 3, 4, 5, 3, 9 };
//vector<int> v{ 1, 2, 3, 4, 5, 3, 9 };



    auto it = find_last2(v, 3);

    cout << *it << endl;
    cout << *(++it) << endl;


    return 0;
}
```

```
3
9
```

STL – Vector

```
template <class T, class A = allocator<T>> class std::vector {
    public:
        // ...

        // element access:
        reference operator[](size_type n);              // unchecked
        const_reference operator[](size_type n) const;

        reference at(size_type n);                      // checked
        const_reference at(size_type n) const;          // throws out_of_range

        reference front();                              // first element
        const_reference front() const;

        reference back();                               // last element
        const_reference back() const;

        // ...
};
```

STL – Vector

```
template <class T, class A = allocator<T>> class std::vector {
    public:
        // ...

        // constructors, etc:
        explicit vector(const A& = A());
        explicit vector(size_type n, const T &val = T(), const A& = A());
        template <class In> vector(In first, In last, const A& = A());
        vector(const vector &x);

        ~vector();

        template <class In> void assign(In first, In last);    // copy
        void assign(size_type n, const T &val);                // n val copies

        // ...
};
```

STL – Vector

```
template <class T, class A = allocator<T>> class std::vector {
    public:
        // ...

        // stack operations:
        void push_back(const T &x);
        void pop_back();

        // list operations:
        iterator insert(interator pos, const T &x);
        void insert(iterator pos, size_type n, const T &x);
        template <class In> void insert(interator pos, In first, In last);

        iterator erase(iterator pos);
        iterator erase(iterator first, iterator last);
        void clear();
        // ...
};
```

STL – Vector

```
template <class T, class A = allocator<T>> class std::vector {
    public:
        // ...
        // capacity:
        size_type size() const;
        bool empty() const { return size() == 0; }
        size_type max_size() const;
        void resize(size_type sz, T val = T());

        size_type capacity() const;
        void reserve(size_type n);

        // other:
        void swap(vector &);
        allocator_type get_allocator() const;

        // ...
};
```

STL – Vector

```
// Helper functions

template <class T, class A>
bool std::operator ==(const vector<T,A> &x, const vector<T,A> &y);

template <class T, class A>
bool std::operator <(const vector<T,A> &x, const vector<T,A> &y) {
    return lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}

template <class T, class A>
void std::swap(vector<T,A> &x, vector<T,A> &y) {
    x.swap(y);
}
```

STL – Vector de bools

- Las variables **`bool`** son addressables, así que ocupan al menos 1 byte.
- La especialización **`vector<bool>`** utiliza 1 bit por elemento para eficiencia.
- Las operaciones de **`vector<bool>`** son las mismas que las de **`vector`**.
- La implementación debe simular el addressing de un bit.
- **`vector<bool>::iterator`** no puede ser un puntero.
- **`vector<bool>::iterator`** no es un **`bool *`**.

STL – Ejemplos de vector

```
void f(vector<int> &v, int i1, int i2)
try {
    for( int i = 0; i < v.size(); i++ ) {
        // range already checked, use unchecked v[i]
    }
    v.at(i1) = v.at(i2); // check range
    // ...
} catch( out_of_range ) {
    // oops: out_of_range error
}

void g(vector<double> &v) {
    double d = v[v.size()];     // undefined, bad index

    list<char> lst;
    char c = lst.front();       // undefined, empty list
}
```

STL – Ejemplos de vector

```
vector<Record> vr(10000);       // cada elemento es inicializado con Record()

void f(int s1, int s2) {
    vector<int> vi(s1);         // cada elemento es inicializado con int() === 0
    vector<double> *p = new vector<double>(s2);
}

class Num {
    public:
        Num(long);    // sin constructor default
        // ...
};

vector<Num> v1(1000);           // error, no hay Num()
vector<Num> v2(1000, Num(0));   // ok

void g(const list<X> &lst) {
    vector<X> v1(lst.begin(), lst.end());
    char p[] = "hola";
    vector<char> v2(p, &p[sizeof(p)-1]);
}
```

STL – Ejemplos de vector

```
vector<int> v1(10);                    // ok: vector de 10 ints
vector<int> v2 = vector<int>(10)       // ok: vector de 10 ints
vector<int> v3 = v2                    // ok: v3 es una copia de v2
vector<int> v4 = 10;                   // error: intenta utilizar una conversión
                                       // implícita de 10 a vector<int>

void f1(vector<int> &);
void f2(const vector<int> &);
void f3(vector<int>);

void h() {
    vector<int> v(10000);

    f1(v);          // pasa por referencia
    f2(v);          // pasa por referencia
    f3(v);          // copia los 10000 elementos a un vector temporario!
}
```

STL – Ejemplos de vector

```
class Book {
    // ...
};

void f(vector<Num> &vn, vector<char> &vc, vector<Book> &vb, list<Book> &lb) {
    vn.assign(10, Num(0));              // asigna 10 copias de Num(0) a vn

    char s[] = "hola";
    vc.assign(s, &s[sizeof(s)-1]);      // asigna "hola" a vc

    vb.assign(lb.begin(), lb.end());
}

void g() {
    vector<char> v(10, 'x');            // v.size() == 10, cada elemento es 'x'
    v.assign(5, 'a');                   // v.size() == 5, cada elemento es 'a'
}
```

STL – Ejemplos de vector

```
vector<Point> cities;

void add_points(Point sentinel) {
    Point buf;

    while( cin >> buf ) {
        if( buff == sentinel )
            return;
        cities.push_back(buf);
    }
}

void f() {
    vector<int> v;
    v.pop_back();        // undefined, v esta vacío
    v.push_back(7);      // undefined, v en estado indefinido
}
```

STL – Ejemplos de vector

```
vector<string> fruit;
fruit.push_back("peach");          fruit.push_back("apple");
fruit.push_back("kiwi");           fruit.push_back("pear");
fruit.push_back("starfruit");      fruit.push_back("grape");

// no me gustan las frutas que empiezan con 'p'
sort(fruit.begin(), fruit.end());

vector<string>::iterator p1 = find_if(fruit.begin(), fruit.end(), initial('p'));
vector<string>::iterator p2 = find_if(p1, fruit.end(), initial_not('p'));
fruit.erase(p1, p2);

// otra manera
vector<string>::iterator p1 = find_if(fruit.begin(), fruit.end(), initial('p'));
vector<string>::reverse_iterator p2 =
                              find_if(fruit.rbegin(), fruit.rend(), initial('p'));
fruit.erase(p1, p2.base());

// chau starfruit
fruit.erase(find(fruit.begin(), fruit.end(), "starfruit"));

fruit.insert(fruit.begin()+1, "cherry");        // inserta después del iterator
fruit.insert(fruit.end(), "cranberry");         // inserta al final
// v.insert(v.end(), x) es equivalente a v.push_back(x)
```

STL – Ejemplos de vector

```
template<class C> void f(C &c)
{
    c.erase(c.begin() + 7);    // ok
    c.erase(&c[7]);            // no es general, no funciona con vector<bool>
    c.erase(c + 7);            // error, c+7 no tiene sentido
    c.erase(c.back());         // error, c.back() es un reference no un iterator
    c.erase(c.end() - 2);      // ok
    c.erase(c.rbegin() + 2);   // error, reverse_iterator no es un iterator

    c.erase((c.rbegin() + 2).base());   // ok, oscuro
}

// c.clear() es equivalente a c.erase(c.begin(), c.end())
```

STL – Ejemplos de vector

```
class Histogram {
        vector<int> count;
    public:
        Histogram(int h) : count(max(h,8)) { }
        void record(int i);
        // ...
};

void Histogram::record(int i)
{
    if( i < 0 )
        i = 0;
    if( count.size() <= i )
        count.resize(i + i);
    count[i]++;
}
```

STL – Ejemplos de vector

```
struct Link {
    Link *next;
    Link(Link *n = 0) : next(n) { }
    // ...
};

vector<Link> v;

void chain(size_t n)
{
    v.reserve(n);
    v.push_back(Link(0));

    for( int i = 1; i < n; i++ )
        v.push_back(Link(&v[i - 1]);
    // ...
}
```

STL – Secuencias y adaptadores

- Las secuencias siguen el pattern de **`vector`**.
- Las secuencias fundamentales son:
  - **`vector`**
  - **`list`**
  - **`deque`**
- A partir de éstas se crean los adaptadores:
  - **`stack`**
  - **`queue`**
  - **`priority_queue`**

STL – List

- `list` prove además de las operaciones generales las siguientes operaciones específicas:

```
template <class T, class A = allocator<T>> class std::list {
    public:
        // list specific operations:
        void splice(iterator pos, list &x); // mueve elts de x a antes de pos
        void splice(iterator pos, list &x, iterator p);  // mueve *p de x
        void splice(iterator pos, list &x, iterator first, iterator last);
        void merge(list &);                                  // merge sorted lists
        template <clas Cmp> void merge(list &, Cmp);
        void sort();
        template <class Cmp> void sort(Cmp);
};
```

STL – List

```
template <class T, class A = allocator<T>> class std::list {
    public:
        // list specific operations:
        reference front();
        const_reference front() const;

        void push_front(const T& val);
        void pop_front();

        void remove(const T& val);
        template <class Pred> void remove_if(Pred p);

        void unique();
        template <class BinPred> void unique(BinPred b);

        void reverse();
};
```

STL – Deque

- **`deque`** es una cola con dos cabezas.
- Está optimizada para que las operaciones en ambas cabezas sean tan eficientes como las de **`list`**, soportando subindexado casi tan eficiente como el de **`vector`**.
- Operaciones de inserción o remoción en el medio son ineficientes como las de **`vector`**.

STL – Stack

```
template <class T, class C = deque<T>> class std::stack {
    protected:
        C c;
    public:
        typedef typename C::value_type value_type;
        typedef typename C::size_type size_type;
        typedef C container_type;
        explicit stack(const C &a = C()) : c(a) { }

        bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }

        value_type &top() { return c.back(); }
        const value_type &top() const { return c.back(); }

        void push(const value_type &x) { c.push_back(x); }
        void pop() { c.pop_back(); }
};
```

STL – Queue

```
template <class T, class C = deque<T>> class std::queue {
    protected:
        C c;

    public:
        typedef typename C::value_type value_type;
        typedef typename C::size_type size_type;
        typedef C container_type;

        explicit queue(const C &a = C()) : c(a) { }

        bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }

        value_type &front() { return c.front(); }
        const value_type &front() const { return c.front(); }

        value_type &back() { return c.back(); }
        const value_type &back() const { return c.back(); }

        void push(const value_type &x) { c.push_back(x); }
        void pop() { c.pop_front(); }
};
```

STL – Priority Queue

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type>>
class std::priority_queue {
    protected:
        C c;
        Cmp cmp;
    public:
        typedef typename C::value_type value_type;
        typedef typename C::size_type size_type;
        typedef C container_type;

        explicit priority queue(const Cmp &a1 = Cmp(), const C &a2 = C())
            : c(a2), cmp(a1) { }
        template<class In>
        priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

        bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }

        const value_type &top() const { return c.front(); }

        void push(const value type &x);
        void pop() { c.pop_front(); }
};
```