Standard Template Library: Contenedores

Estructuras de Datos

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Índice de contenido

Standard Template Library: Contenedores

Un contenedor es un objeto que almacena una colección de otros objetos (denominados sus elementos). Se implementan como clases plantilla lo que permite una mayor flexibilidad en los tipos usados por sus elementos.
El contenedor se dedica a organizar el espacio para sus elementos y producir una interfaz a través de sus funciones miembro para acceder a estos elementos, de una manera directa o a través del uso de iteradores (objetos que direccionan a los elementos del contenedor de una forma parecida a los punteros).

Los contenedores replican estructuras muy parecidas a las usadas normalmente en programación: vector dinámico (vector), colas (queue), pilas (stack), colas con prioridad (priority_queue), listas enlazadas (list), arboles (set), vectores asociativos (map)... Muchos de los contenedores tienen diferentes funciones miembro en común, y comparten la funcionalidad. La decisión sobre que contenedores usar en nuestra aplicación no depende solamente sobre la funcionalidad ofrecida por el contenedor sino también por la eficiencia de alguno de sus funciones miembros. Por ejemplo este hecho es muy visible para los contenedores secuenciales (vectores, colas, etc) donde las operaciones de insercción, borrado y consulta suele ser un punto donde plantearse que eficiencia computacional presenta estas funciones para elegir estos tipos de contenedores.

stack, queue and priority_queue se implementan como contendores adaptados (en inglés *container adaptors)*. Container adaptors no son clases contendores por si solas, ya que estas clases dan una interfaz sobre clases cotenedores sobre las que se representan.

Clases plantillas de Contendores

Contenedores

vector Vector (class template)

list Lista(class template)

Contenedores basados en otros

stack Pila (class template)

queue Cola(class template)

priority_queue Cola con prioridad(class template)

Contenedores Asociativos

set Conjunto(class template)

multiset Conjunto con multiples claves (class template)

multimap Arboles con múltiples claves (class template)

Funciones Miembro

			Contenedores	
Cabeceras			**<vector>**	**<list>**
Funciones Miembro		Eficiencia	**vector**	list
	constructor	*	constructor	constructor
	destructor	O(n)	destructor	destructor
	operator=	O(n)	operator=	operator=
iteradores	begin	O(1)	begin	begin
	end	O(1)	end	end
	rbegin	O(1)	rbegin	rbegin
	rend	O(1)	rend	rend
capacidad	size	*	size	size
	max_size	*	max_size	max_size
	empty	O(1)	empty	empty
	resize	O(n)	resize	resize
acceso a los elementos	front	O(1)	front	front
	back	O(1)	back	back
	operator[]	*	operator[]	
	at	O(1)	at	
modificadores	assign	O(n)	assign	assign
	insert	*	insert	insert
	erase	*	erase	erase
	swap	O(1)	swap	swap
	clear	O(n)	clear	clear
	push_front	O(1)		push_front
	pop_front	O(1)		pop_front
	push_back	O(1)	push_back	push_back
	pop_back	O(1)	pop_back	pop_back

observadores	`key_comp`	O(1)		
	`value_comp`	O(1)		
operaciones	`find`	O(log n)		
	`count`	O(log n)		
	`lower_bound`	O(log n)		
	`upper_bound`	O(log n)		
	`equal_range`	O(log n)		
miembros únicos			capacity reserve	splice remove remove_if unique merge sort reverse

O(1) constante < O(log n) logaritmica < O(n) lineal; *=depende del contendor

			Contenedores Asociativos			
Cabeceras			**<set>**		**<map>**	
Funciones Miembro		Eficiencia	**set**	**multiset**	**map**	**multimap**
	constructor	*	constructor	constructor	constructor	constructor
	destructor	O(n)	destructor	destructor	destructor	destructor
	operator=	O(n)	operator=	operator=	operator=	operator=
iteradores	begin	O(1)	begin	begin	begin	begin
	end	O(1)	end	end	end	end
	rbegin	O(1)	rbegin	rbegin	rbegin	rbegin
	rend	O(1)	rend	rend	rend	rend
capacidad	size	*	size	size	size	size
	max_size	*	max_size	max_size	max_size	max_size
	empty	O(1)	empty	empty	empty	empty
	resize	O(n)				
acceso a elementos	front	O(1)				
	back	O(1)				
	operator[]	*			operator[]	
	at	O(1)				
modificadores	assign	O(n)				
	insert	*	insert	insert	insert	insert
	erase	*	erase	erase	erase	erase
	swap	O(1)	swap	swap	swap	swap
	clear	O(n)	clear	clear	clear	clear
	push_front	O(1)				
	pop_front	O(1)				
	push_back	O(1)				
	pop_back	O(1)				
observadores	key_comp	O(1)	key_comp	key_comp	key_comp	key_comp

	`value_comp`	O(1)	value_comp	value_comp	value_comp	value_comp
operaciones	`find`	O(log n)	find	find	find	find
	`count`	O(log n)	count	count	count	count
	`lower_bound`	O(log n)	lower_bound	lower_bound	lower_bound	lower_bound
	`upper_bound`	O(log n)	upper_bound	upper_bound	upper_bound	upper_bound
	`equal_range`	O(log n)	equal_range	equal_range	equal_range	equal_range
Miembros únicos						

O(1) constante< O(log n) logaritmica < O(n) lineal; *=depende del contendor

			Contenedores Adaptados		
Cabeceras			**<stack>**	**<queue>**	
Funciones Miembro			**stack**	**queue**	**priority_queue**
	constructor	*	constructor	constructor	constructor
capacidad	`size`	O(1)	size	size	size
	`empty`	O(1)	empty	empty	empty
acceso a elemento	`front`	O(1)		front	
	`back`	O(1)		back	
	`top`	O(1)	top		top
modificadores	`push`	O(1)	push	push	push
	`pop`	O(1)	pop	pop	pop

VECTOR

class template
<vector>

1.1. vector::vector (Constructor)

Contruye un cotenedor vector. Según la inicialización del vector elegida así será llamado al constructor correspondiente. Estos son:

- explicit vector (const Allocator& = Allocator());

  Contructor por defecto:construye un `vector` vacío de tamaño cero
- explicit vector (size_type n, const T& value= T(), const Allocator& = Allocator());

  Contructor para duplicar una secuencia:Inicializa el vector de dimension n con todos los valores inicializados a *value*
- template <class InputIterator> vector (InputIterator first, InputIterator last, const Allocator& = Allocator());

  Contructor por iteracion: Itera entre *first* and *last*, poniendo una copia de cada uno de los elementos de la secuencia como el contenido de los elementos del vector.
- vector (const vector<T,Allocator>& x);
  Contructor de Copia: El vector se inicializa para tener los mismos contenidos y propiedades del vecor x

Ejemplo

```
1  // constructing vectors
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main ()
7  {
8    unsigned int i;
9
10   // constructors used in the same order as described above:
11   vector<int> first;                                // empty vector of ints
12   vector<int> second (4,100);                       // four ints with value 100
13   vector<int> third (second.begin(),second.end());  // iterating through second
14   vector<int> fourth (third);                       // a copy of third
15
16   // the iterator constructor can also be used to construct from arrays:
17   int myints[] = {16,2,77,29};
18   vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
19
     cout << "The contents of fifth are:";
```

```
    for (i=0; i < fifth.size(); i++)
20    cout << " " << fifth[i];
21
22  cout << endl;
23
24  return 0;
25 }
```

1.2. vector::~vector (Destructor)

Destruye el contenedor. Desde esta función se llama a los destructores de cada uno de los elementos en el vector.

1.3. vector::operator=

Asigna una copia del vector x como el nuevo cotenido del vector.
Los elementos que contenía el vector antes de la llamada son eliminados y sustituido por copias de los elementos del vector x.
Después de la llamada ambos el vector y x tendrán el mismo número de elementos y contendrán los mismos elementos.

Ejemplo

```
   // vector assignment
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> first (3,0);
8    vector<int> second (5,0);
9
10   second=first;
11   first=vector<int>();
12
13   cout << "Size of first: " << int (first.size()) << endl;
14   cout << "Size of second: " << int (second.size()) << endl;
15   return 0;
16 }
17
```

1.4. vector::begin

```
     iterator begin ();
const_iterator begin () const;
```

Devuelve un iterador que apunta al primer elemento del vector.

Ambos `iterator` y `const_iterator` son tipos miembro de vector.

Ejemplo:

```
   // vector::begin
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> myvector;
8    for (int i=1; i<=5; i++) myvector.push_back(i);
9
10   vector<int>::iterator it;
11
12   cout << "myvector contains:";
13   for ( it=myvector.begin() ; it < myvector.end(); it++ )
14     cout << " " << *it;
15
16   cout << endl;
17
18   return 0;
19 }
20
```

1.5. vector::end

```
      iterator end ();
const_iterator end () const;
```

Devuelve un iterador apuntando al elemento trás el último en el vector. An iterator to the element past the end of the sequence.

Ejemplo

```
   // vector::end
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> myvector;
8    for (int i=1; i<=5; i++) myvector.insert(myvector.end(),i);
9
10   cout << "myvector contains:";
11   vector<int>::iterator it;
12   for ( it=myvector.begin() ; it < myvector.end(); it++ )
13     cout << " " << *it;
14
15   cout << endl;
16
17   return 0;
   }
```

1.6. vector::rbegin

```
      reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

Devuelve un iterador apuntado al último elemento en el contenedor. `rbegin` apunta al elemento último (más a la derecha del vector)

Un iterador inverso (reverse) comienza desde el último elemento del vector al primero

Ejemplo

```
   // vector::rbegin/rend
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> myvector;
8    for (int i=1; i<=5; i++) myvector.push_back(i);
9
10   cout << "myvector contains:";
11   vector<int>::reverse_iterator rit;
12   for ( rit=myvector.rbegin() ; rit < myvector.rend(); ++rit )
13     cout << " " << *rit;
14
15   cout << endl;
16
17   return 0;
18 }
19
```

1.7. vector::rend

```
      reverse_iterator rend();
const_reverse_iterator rend() const;
```

Devuelve un iterador inverso apuntando al elemento previo al primer elemento del vector.

Ejemplo

```
1  // vector::rbegin/rend
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main ()
7  {
8    vector<int> myvector;
9    for (int i=1; i<=5; i++) myvector.push_back(i);
10
11   cout << "myvector contains:";
12   vector<int>::reverse_iterator rit;
13   for ( rit=myvector.rbegin() ; rit < myvector.rend(); ++rit )
```

```
      cout << " " << *rit;
14
15    cout << endl;
16
17    return 0;
18  }
19
```

1.8. vector::size

```
size_type size() const;
```

Devuelve el número del elementos del vector . Hay que distinguir que el número de elementos del vector no tiene por que ser igual a la capacidad del vector. Los vectores automáticamente reasignan espacio cuando sea necesario o cuando se llama a la función resize.

El tipo `size_type` is an unsigned integral.

Ejemplo

```
    // vector::size
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main ()
6   {
7     vector<int> myints;
8     cout << "0. size: " << (int) myints.size() << endl;
9
10    for (int i=0; i<10; i++) myints.push_back(i);
11    cout << "1. size: " << (int) myints.size() << endl;
12
13    myints.insert (myints.end(),10,100);
14    cout << "2. size: " << (int) myints.size() << endl;
15
16    myints.pop_back();
17    cout << "3. size: " << (int) myints.size() << endl;
18
19    return 0;
20  }
21
```

1.9. vector::max_size

```
size_type max_size () const;
```

Devuelve el número máximo de elementos que puede contener el vector.
El tamaño potencial máximo que puede contener el vector está en función del sistema o de las limitaciones de la librería que se está usando.

Ejemplo

```
   // comparing size, capacity and max_size
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> myvector;
8
9    // set some content in the vector:
10   for (int i=0; i<100; i++) myvector.push_back(i);
11
12   cout << "size: " << myvector.size() << "\n";
13   cout << "capacity: " << myvector.capacity() << "\n";
14   cout << "max_size: " << myvector.max_size() << "\n";
15   return 0;
16 }
17
```

1.10. vector::resize

```
void resize ( size_type sz, T c = T() );
```

Cambia el tamaño del vector a *sz* elementos.
Si sz es menor que el tamaño actual del vector (size), el contenedor se reduce a sus primeros *sz* elementos, los otros son eliminados.
Si *sz* es mayor que el tamaño actual del vector (size), el contenido se expande insertando al final tantas copias de c hasta completar sz elementos. Puede causar una reasignación de espacio.

Ejemplo

```
   // resizing vector
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> myvector;
8
9    unsigned int i;
10
11   // set some initial content:
12   for (i=1;i<10;i++) myvector.push_back(i);
13
14   myvector.resize(5);
15   myvector.resize(8,100);
16   myvector.resize(12);
17
18   cout << "myvector contains:";
19   for (i=0;i<myvector.size();i++)
20     cout << " " << myvector[i];
21
22   cout << endl;
23
```

```
24    return 0;
25  }
26
```

1.11. vector::front

```
      reference front ( );
const_reference front ( ) const;
```

Devuelve una referencia al primer elemento del vector, a diferencia de vector::begin que devuelve un iterador al primer elemento.

Ejemplo

```
    // vector::front
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main ()
6   {
7     vector<int> myvector;
8
9     myvector.push_back(78);
10    myvector.push_back(16);
11
12    // now front equals 78, and back 16
13
14    myvector.front() -= myvector.back();
15
16    cout << "myvector.front() is now " << myvector.front() << endl;
17
18    return 0;
19  }
```

1.12. vector::back

```
      reference back ( );
const_reference back ( ) const;
```

Devuelve una referencia al último elemento del vector, a diferencia de vector::end que devuelve un iterador trás el último elemento.

Ejemplo

```
    // vector::back
1   #include <iostream>
2   #include <vector>
3   using namespace std;
```

```
4
5   int main ()
6   {
7     vector<int> myvector;
8
9     myvector.push_back(10);
10
11    while (myvector.back() != 0)
12    {
13      myvector.push_back ( myvector.back() -1 );
14    }
15
16    cout << "myvector contains:";
17    for (unsigned i=0; i<myvector.size() ; i++)
18      cout << " " << myvector[i];
19
20    cout << endl;
21
22    return 0;
23
```

1.13. vector::operator[]

```
      reference operator[] ( size_type n );
const_reference operator[] ( size_type n ) const;
```

Accede a un elemento. Devuelve una referencia al elemento en la posición n. Similar a la función vector::at excepto que vector::at lanza una excepción si la posición está fuera de rango.

Ejemplo

```
    // vector::operator[]
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main ()
6   {
7     vector<int> myvector (10);   // 10 zero-initialized elements
8     unsigned int i;
9
10    vector<int>::size_type sz = myvector.size();
11
12    // assign some values:
13    for (i=0; i<sz; i++) myvector[i]=i;
14
15    // reverse vector using operator[]:
16    for (i=0; i<sz/2; i++)
17    {
18      int temp;
19      temp = myvector[sz-1-i];
20      myvector[sz-1-i]=myvector[i];
21      myvector[i]=temp;
```

```
    }

  cout << "myvector contains:";
  for (i=0; i<sz; i++)
    cout << " " << myvector[i];

  cout << endl;

  return 0;
}
```

1.14. vector::at

```
const_reference at ( size_type n ) const;
      reference at ( size_type n );
```

Devuelve una referencia al elemento en la posicion n del vector. Lanza una excepción si n está fuera de rango.

Ejemplo

```
   // vector::at
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> myvector (10);   // 10 zero-initialized ints
8    unsigned int i;
9
10   // assign some values:
11   for (i=0; i<myvector.size(); i++)
12     myvector.at(i)=i;
13
14   cout << "myvector contains:";
15   for (i=0; i<myvector.size(); i++)
16     cout << " " << myvector.at(i);
17
18   cout << endl;
19
20   return 0;
   }
```

1.15. vector::assign

```
template <class InputIterator>
  void assign ( InputIterator first, InputIterator last );
void assign ( size_type n, const T& u );
```

Asigna un contenido nuevo al vector elmininando todos los elementos que pudiera tener

antes de la llamada.
En la primera version (con iteradores) el nuevo contenido son una copia de todos los elementos en la secuencia que va desde first hasta last.
En la segunda versión, el vector almacena n copias del elemento u.

Ejemplo

```
   // vector assign
   #include <iostream>
   #include <vector>
1  using namespace std;
2
3  int main ()
4  {
5    vector<int> first;
6    vector<int> second;
7    vector<int> third;
8
9    first.assign (7,100);             // a repetition 7 times of value 100
10
11   vector<int>::iterator it;
12   it=first.begin()+1;
13
14   second.assign (it,first.end()-1); // the 5 central values of first
15
16   int myints[] = {1776,7,4};
17   third.assign (myints,myints+3);   // assigning from array.
18
19   cout << "Size of first: " << int (first.size()) << endl;
20   cout << "Size of second: " << int (second.size()) << endl;
     cout << "Size of third: " << int (third.size()) << endl;
     return 0;}
```

1.16. vector::insert

```
iterator insert ( iterator position, const T& x );
    void insert ( iterator position, size_type n, const T& x );
template <class InputIterator>
    void insert ( iterator position, InputIterator first, InputIterator last );
```

Se inserta nuevos elementos en el vector antes del elemento en *position.* Se incrementa el tamaño del vector, reasignando el espacio que ocupa si el tamaño del nuevo vector supera la capacidad del nuevo vector. Si se reasigna nuevo espacio todos los iteradores previos quedan invalidados.
Las inserciones previas a end son realizadas moviendo todos los elementos entre position y end con el objetivo de insertar los nuevos elementos.

Ejemplo

```
   // inserting into a vector
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
```

```
7     vector<int> myvector (3,100);
8     vector<int>::iterator it;
9
10    it = myvector.begin();
11    it = myvector.insert ( it , 200 );
12
13    myvector.insert (it,2,300);
14
15    // "it" no longer valid, get a new one:
16    it = myvector.begin();
17
18    vector<int> anothervector (2,400);
19    myvector.insert (it+2,anothervector.begin(),anothervector.end());
20
21    int myarray [] = { 501,502,503 };
22    myvector.insert (myvector.begin(), myarray, myarray+3);
23
24    cout << "myvector contains:";
25    for (it=myvector.begin(); it<myvector.end(); it++)
26      cout << " " << *it;
27    cout << endl;
28
29    return 0;
30  }
31
```

1.17. vector ::erase

```
iterator erase ( iterator position );
iterator erase ( iterator first, iterator last );
```

Elimina un elemento o varios: el que se encuentra en *position* o en el rango que va desde first hasta last. Se reduce el tamaño del vector por el número de elementos borrados (se invoca al destructor de cada uno de los elementos).

Devuelve un iterador al elemento siguiente a position o a last.

Ejemplo

```
1   // erasing from vector
2   #include <iostream>
3   #include <vector>
4   using namespace std;
5
6   int main ()
7   {
8     unsigned int i;
9     vector<unsigned int> myvector;
10
11    // set some values (from 1 to 10)
12    for (i=1; i<=10; i++) myvector.push_back(i);
13
14    // erase the 6th element
15    myvector.erase (myvector.begin()+5);
16
      // erase the first 3 elements:
```

```
17    myvector.erase (myvector.begin(),myvector.begin()+3);
18
19    cout << "myvector contains:";
20    for (i=0; i<myvector.size(); i++)
21      cout << " " << myvector[i];
22    cout << endl;
23
24    return 0;
25  }
26
```

1.18. vector::swap

```
void swap ( vector<T,Allocator>& vec );
```

Intercambia el contenido del vector por el contenido de vec, que es otro vector del mismo tipos. El tamaño puede ser diferente. Así después de la llamada los elementos del vector son los que tenía vec y los de vec son los que tenía el vector.

Note que la funcion swap en la biblioteca algorithm existe con el mismo nombre y comportamiento.

Ejemplo

```
    // swap vectors
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   main ()
6   {
7     unsigned int i;
8     vector<int> first (3,100);   // three ints with a value of 100
9     vector<int> second (5,200);  // five ints with a value of 200
10
11    first.swap(second);
12
13    cout << "first contains:";
14    for (i=0; i<first.size(); i++) cout << " " << first[i];
15
16    cout << "\nsecond contains:";
17    for (i=0; i<second.size(); i++) cout << " " << second[i];
18
19    cout << endl;
20
21    return 0;
22  }
23
```

1.19. vector ::clear

```
void clear ( );
```

Clear content

Se borran todos los elementos del vector: se llaman a sus destructores y son eliminados del vector, dejando el contenedor con tamaño 0.

Ejemplo

```
   // clearing vectors
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    unsigned int i;
8    vector<int> myvector;
9    myvector.push_back (100);
10   myvector.push_back (200);
11   myvector.push_back (300);
12
13   cout << "myvector contains:";
14   for (i=0; i<myvector.size(); i++) cout << " " << myvector[i];
15
16   myvector.clear();
17   myvector.push_back (1101);
18   myvector.push_back (2202);
19
20   cout << "\nmyvector contains:";
21   for (i=0; i<myvector.size(); i++) cout << " " << myvector[i];
22
23   cout << endl;
24
25   return 0;
26 }
27
```

1.20. vector ::push_back

```
void push_back ( const T& x );
```

Añade un nuevo elemento al vector, siendo este el último elemento del vector. Incrementa el tamaño (size) del vector por 1, pudiendo causar una reasignación de memoria si hiciese falta. Si se hace una reasignación de memoria todos los iteradores previos quedan invalidados.

Ejemplo

```
   // vector::push_back
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> myvector;
8    int myint;
9
10   cout << "Please enter some integers (enter 0 to end):\n";
11
12
```

```
    do {
      cin >> myint;
13    myvector.push_back (myint);
14  } while (myint);
15
16  cout << "myvector stores " << (int) myvector.size() << " numbers.\n";

    return 0;}
```

1.21. vector ::pop_back

```
void pop_back ( );
```

Borra el último elemento del vector, reduciendo en uno el tamaño (size) del vector. Invalida todos los iteradores que apuntaban a este elemento.

Ejemplo

1.22. vector::capacity

```
size_type capacity () const;
```

Devuelve el tamaño del espacio asignado al vector. La capacidad no tiene porque ser igual al número de elementos pero si tiene que ser obligatoriamente mayor o igual al numero de elementos (size).

La capacidad no es un límite del espacio asignado al vector ya que si el vector necesita más espacio para ubicar más elementos la capacidad será ampliada. El límite del espacio asignado al vector lo da la funcion max_size (que depende del sistema o librería).

Ejemplo

```
1   // comparing size, capacity and max_size
2   #include <iostream>
3   #include <vector>
4   using namespace std;
5
6   int main ()
7   {
8     vector<int> myvector;
9
10    // set some content in the vector:
11    for (int i=0; i<100; i++) myvector.push_back(i);
12
13    cout << "size: " << (int) myvector.size() << "\n";
14    cout << "capacity: " << (int) myvector.capacity() << "\n";
15    cout << "max_size: " << (int) myvector.max_size() << "\n";
16    return 0;
17
```

1.23. vector ::reserve

```
void reserve ( size_type n );
```

Petición de cambiar la capacidad asignada al vector, solicitando que al menos necesita n elementos. El parámetro n indica un mínimo de capacidad que se requiere así que la dada puede ser mayor o igual que n.

Cuand n es mayor que la capacidad actual, se realiza una reasignacón de memoria durante la llamada a esta funcion. La reasignación de memoria invalida todos los iteradores previos.

Ejemplo

```
   // vector::reserve
   #include <iostream>
1  #include <fstream>
2  #include <vector>
3  using namespace std;
4
5  int main ()
6  {
7    vector<int> content;
8    size_t filesize;
9
10   ifstream file ("test.bin",ios::in|ios::ate|ios::binary);
11   if (file.is_open())
12   {
13     filesize=file.tellg();
14
15     content.reserve(filesize);
16
17     file.seekg(0);
18     while (!file.eof())
19     {
20       content.push_back( file.get() );
21     }
22
23     // print out content:
24     vector<int>::iterator it;
25     for (it=content.begin() ; it<content.end() ; it++)
26       cout << hex << *it;
27   }

     return 0;}
```

List (Listas)

class template

<list>

1.24. list::list

```
explicit list ( const Allocator& = Allocator() );
explicit list ( size_type n, const T& value = T(), const Allocator& =
Allocator() );
template < class InputIterator >
           list ( InputIterator first, InputIterator last, const Allocator& =
Allocator() );
list ( const list<T,Allocator>& x );
```

Construye un contenedor list, inicializando sus contenidos dependiendo de la versión del constructor que se use:

- Contructor por defecto: construye una lista vacía, y el tamaño es cero.
- Constructor repitiendo el valor: Inicializa el contendor con elementos repitiendo valor en cada uno de ellos.
- Contructor iteración: Itera entre first y last poniendo un copia de cada uno de los elementos en el contendor
- Contructor de Copia: El objeto se inicializa para tener el mismo contenido y propiedades de la lista x.

Ejemplo

```
1  // constructing lists
2  #include <iostream>
3  #include <list>
4  using namespace std;
5
6  int main ()
7  {
8    // constructors used in the same order as described above:
9    list<int> first;                                // empty list of ints
10   list<int> second (4,100);                       // four ints with value 100
11   list<int> third (second.begin(),second.end());  // iterating through second
12   list<int> fourth (third);                       // a copy of third
13
14   // the iterator constructor can also be used to construct from arrays:
15   int myints[] = {16,2,77,29};
16   list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
17
18   cout << "The contents of fifth are: ";
19   for (list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
       cout << *it << " ";

     cout << endl;return 0;}
```

1.25. list ::~list

```
~list ( );
```

Destruye el contenedor. Llama a cada uno de los destructore del contenedor.

1.26. list::operator=

```
list<T,Allocator>& operator= ( const list<T,Allocator>& x );
```

Asigna como el nuevo contenido para el contenedor una copia de los elementos de x. Los elementos previos del objeto son eliminados, y sustituidos por copias de aquellos en el vector x.
Despues de la llamada a esta función, ambas listas tiene los mismos elementos y comparado elemento a elemento son iguales.

Ejemplo

```
   // assignment operator with lists
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> first (3);      // list of 3 zero-initialized ints
8    list<int> second (5);     // list of 5 zero-initialized ints
9
10   second=first;
11   first=list<int>();
12
13   cout << "Size of first: " << int (first.size()) << endl;
14   cout << "Size of second: " << int (second.size()) << endl;
15   return 0;
16 }
17
```

1.27. list::begin

```
      iterator begin ();
const_iterator begin () const;
```

Devuelve un iterador apuntando al primer elemento de la lista.

Ejemplo

```
1  // list::begin
2  #include <iostream>
3  #include <list>
4  using namespace std;
5
```

```
   int main ()
6  {
7    int myints[] = {75,23,65,42,13};
8    list<int> mylist (myints,myints+5);
9
10   list<int>::iterator it;
11
12   cout << "mylist contains:";
13   for ( it=mylist.begin() ; it != mylist.end(); it++ )
14     cout << " " << *it;
15
16   cout << endl;
17
18   return 0;
19 }
20
```

1.28. list::end

```
      iterator end ();
const_iterator end () const;
```

Devuelve un iterador apuntando al elemento trás el último de la lista.

Ejemplo

```
1  // list::begin/end
2  #include <iostream>
3  #include <list>
4  using namespace std;
5
6  int main ()
7  {
8    int myints[] = {75,23,65,42,13};
9    list<int> mylist (myints,myints+5);
10
11   list<int>::iterator it;
12
     cout << "mylist contains:";
     for ( it=mylist.begin() ; it != mylist.end(); it++ )
13     cout << " " << *it;
14
15   cout << endl;
16
17   return 0;
   }
```

1.29. list::rbegin

```
      reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

Devuelve un iterador inverso apuntando al último elemento de la lista. `rbegin` refiere al elemento a la derecha antes del que apunta la función miembro end.

Ejemplo

```
   // list::rbegin/rend
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
8    for (int i=1; i<=5; i++) mylist.push_back(i);
9
10   cout << "mylist contains:";
11   list<int>::reverse_iterator rit;
12   for ( rit=mylist.rbegin() ; rit != mylist.rend(); ++rit )
13     cout << " " << *rit;
14
15   cout << endl;
16
17   return 0;
18 }
19
```

1.30. list::rend

```
      reverse_iterator rend();
const_reverse_iterator rend() const;
```

Devuelve un iterador inverso apuntando al elemento antes del primero elemento

Ejemplo

```
   // list::rbegin/rend
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
8    for (int i=1; i<=5; i++) mylist.push_back(i);
9
10   cout << "mylist contains:";
11   list<int>::reverse_iterator rit;
12   for ( rit=mylist.rbegin() ; rit != mylist.rend(); ++rit )
13     cout << " " << *rit;
14
15   cout << endl;
```

```
16
17    return 0;
18  }
19
```

1.31. list::size

```
size_type size() const;
```

Devuelve el número de elemento en el contendor

Ejemplo

```
    // list::size
1   #include <iostream>
2   #include <list>
3   using namespace std;
4
5   int main ()
6   {
7     list<int> myints;
8     cout << "0. size: " << (int) myints.size() << endl;
9
10    for (int i=0; i<10; i++) myints.push_back(i);
11    cout << "1. size: " << (int) myints.size() << endl;
12
13    myints.insert (myints.begin(),10,100);
14    cout << "2. size: " << (int) myints.size() << endl;
15
16    myints.pop_back();
17    cout << "3. size: " << (int) myints.size() << endl;
18
19    return 0;
20  }
21
```

1.32. list::max_size

```
size_type max_size () const;
```

Devuelve el número máximo de elementos que puede contener la lista. Este máximo depende del sistema o las limitaciones de la implementación.

Ejemplo

```
    // list::max_size
1   #include <iostream>
2   #include <list>
3   using namespace std;
4
5   int main ()
6   {
7     unsigned int i;
8
```

```
    list<int> mylist;
9
10   cout << "Enter number of elements: ";
11   cin >> i;
12
13   if (i<mylist.max_size()) mylist.resize(i);
14   else cout << "That size exceeds the limit.\n";
15
16   return 0;
17 }
18
```

1.33. list::empty

```
bool empty ( ) const;
```

Devuelve true si la lista esta vacía (tamaño 0).

Ejemplo

```
   // list::empty
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
8    int sum (0);
9
10   for (int i=1;i<=10;i++) mylist.push_back(i);
11
12   while (!mylist.empty())
13   {
14      sum += mylist.front();
15      mylist.pop_front();
16   }
17
18   cout << "total: " << sum << endl;
19
20   return 0;
21 }
22
```

1.34. list::resize

```
void resize ( size_type sz, T c = T() );
```

Cambia el tamaño del contenedor a sz elementos.
Si sz es menor que el tamaño del actual contenedor, el contenido se reduce a sus sz primeros elementos el resto son eliminados.

Si sz es mayor que el actual tamaño, el contenido se expande insertando al final tantas copias de *c* con se necesiten hasta llegar a sz elementos.
Esta función cambia el contenido actual del contenedor bien eliminando o insertando elementos a él.

Ejemplo

```
1   // resizing list
2   #include <iostream>
3   #include <list>
4   using namespace std;
5
6   int main ()
7   {
8     list<int> mylist;
9
10    unsigned int i;
11
12    // set some initial content:
13    for (i=1;i<10;i++) mylist.push_back(i);
14
15    mylist.resize(5);
16    mylist.resize(8,100);
17    mylist.resize(12);
18
19    cout << "mylist contains:";
20    for (list<int>::iterator it=mylist.begin();it!=mylist.end();++it)
21      cout << " " << *it;
22
23    cout << endl;

      return 0;
    }
```

1.35. list:front

```
      reference front ( );
const_reference front ( ) const;
```

Devuelve una referencia al primer elemento la lista.

Ejemplo

```
1   // list::front
2   #include <iostream>
3   #include <list>
4   using namespace std;
5
6   int main ()
7   {
8     list<int> mylist;
9
10    mylist.push_back(77);
11    mylist.push_back(16);
12
13    // now front equals 77, and back 16
14
      mylist.front() -= mylist.back();
```

```
15
16    cout << "mylist.front() is now " << mylist.front() << endl;
17
18    return 0;
19  }
20
```

1.36. list::back

```
      reference back ( );
const_reference back ( ) const;
```

Devuelve una referencia al último elemento en el contenedor

Ejemplo

```
    // list::back
1   #include <iostream>
2   #include <list>
3   using namespace std;
4
5   int main ()
6   {
7     list<int> mylist;
8
9     mylist.push_back(10);
10
11    while (mylist.back() != 0)
12    {
13      mylist.push_back ( mylist.back() -1 );
14    }
15
16    cout << "mylist contains:";
17    for (list<int>::iterator it=mylist.begin(); it!=mylist.end() ; ++it)
18      cout << " " << *it;
19
20    cout << endl;
21
22    return 0;
23  }
24
```

1.37. list::assign

```
template <class InputIterator>
  void assign ( InputIterator first, InputIterator last );
void assign ( size_type n, const T& u );
```

Asigna un nuevo contenido al contendor, eliminando todos los elementos que hubiera previos a la llamada de la funcion miembro y sustituyéndolos por aquellos que se especifican en los parámetros:
En la primera verisón (con iteradores), el nuevo contenido del contenedor es una copia a aquellos contenidos en ls secuencia entre first and last. En la segunda versión, el nuevo contenido es la repetición de n veces del elemento u.

Ejemplo:

```
   // list::assign
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> first;
8    list<int> second;
9
10   first.assign (7,100);                      // 7 ints with value 100
11
12   second.assign (first.begin(),first.end()); // a copy of first
13
14   int myints[]={1776,7,4};
15   first.assign (myints,myints+3);            // assigning from array
16
17   cout << "Size of first: " << int (first.size()) << endl;
18   cout << "Size of second: " << int (second.size()) << endl;
19   return 0;
20 }
21
```

1.38. list:insert

```
iterator insert ( iterator position, const T& x );
    void insert ( iterator position, size_type n, const T& x );
template <class InputIterator>
    void insert ( iterator position, InputIterator first, InputIterator last );
```

La lista se amplía con la insercción de nuevos elementos antes de la posicion position. Se incrementa el tamaño del contendor por la cantidad de elementos insertados. Lista es el tipo de contenedor haciendo de una forma más eficiente, que p.e vector, las operaciones de insercción

Ejemplo

```
   // inserting into a list
1  #include <iostream>
2  #include <list>
3  #include <vector>
4  using namespace std;
5
6  int main ()
7  {
8    list<int> mylist;
9    list<int>::iterator it;
10
11   // set some initial values:
12   for (int i=1; i<=5; i++) mylist.push_back(i); // 1 2 3 4 5
13
14
```

```
    it = mylist.begin();
15  ++it;       // it points now to number 2           ^
16
17  mylist.insert (it,10);                        // 1 10 2 3 4 5
18
19  // "it" still points to number 2                     ^
20  mylist.insert (it,2,20);                      // 1 10 20 20 2 3 4 5
21
22  --it;       // it points now to the second 20            ^
23
24  vector<int> myvector (2,30);
25  mylist.insert (it,myvector.begin(),myvector.end());
26                                                // 1 10 20 30 30 20 2 3 4 5
27                                                //               ^
28  cout << "mylist contains:";
29  for (it=mylist.begin(); it!=mylist.end(); it++)
30    cout << " " << *it;
31  cout << endl;
32
33  return 0;
34 }
35
```

1.39. list::erase

```
iterator erase ( iterator position );
iterator erase ( iterator first, iterator last );
```

Elimina uno o varios elementos de la lista. Elimina el elemento dado en position o elimina el rango de elementos entre first and last.
Se reduce el tamaño de la lista al eliminar elementos.Para eliminar dichos elementos se llama al destructor de cada uno de ellos.

Ejemplo

```
1   // erasing from list
2   #include <iostream>
3   #include <list>
4   using namespace std;
5
6   int main ()
7   {
8     unsigned int i;
9     list<unsigned int> mylist;
10    list<unsigned int>::iterator it1,it2;
11
12    // set some values:
13    for (i=1; i<10; i++) mylist.push_back(i*10);
14
15                               // 10 20 30 40 50 60 70 80 90
16    it1 = it2 = mylist.begin(); // ^^
17    advance (it2,6);           // ^                 ^
18    ++it1;                     //    ^              ^
```

```
    it1 = mylist.erase (it1);   // 10 30 40 50 60 70 80 90
                                //    ^           ^
19
20  it2 = mylist.erase (it2);   // 10 30 40 50 60 80 90
21                              //    ^           ^
22
23  ++it1;                      //       ^        ^
24  --it2;                      //       ^     ^
25
26  mylist.erase (it1,it2);     // 10 30 60 80 90
27                              //        ^
28
29  cout << "mylist contains:";
30  for (it1=mylist.begin(); it1!=mylist.end(); ++it1)
31    cout << " " << *it1;
    cout << endl;

    return 0;}
```

1.40. list::swap

```
void swap ( list<T,Allocator>& lst );
```

Intercambia el contenido del vector por el cotenido de lst, que contiene elementos del mismo tipo. El tamaño puede ser diferente. Después de la llamada a esta función miembro, los elementos de este contenedor son aquellos que estaban en lst antes de la llamada, y los elementos de lst son los que estaban en el vector. Todos los iteradores, referencias y punteros permanecen válidos despues del intercambio.

Ejemplo

```
   // swap lists
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> first (3,100);   // three ints with a value of 100
8    list<int> second (5,200);  // five ints with a value of 200
9    list<int>::iterator it;
10
11   first.swap(second);
12
13   cout << "first contains:";
14   for (it=first.begin(); it!=first.end(); it++) cout << " " << *it;
15
16   cout << "\nsecond contains:";
17   for (it=second.begin(); it!=second.end(); it++) cout << " " << *it;
18
19   cout << endl;
20
21   return 0;
22 }
23
```

1.41. list::clear

```
void clear ( );
```

Todos los elementos de list son eliminados: se llaman al destructor de cada elemento y a continuación son eliminados de la lista dejándola con un tamaño de 0.

Ejemplo

```
   // clearing lists
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
8    list<int>::iterator it;
9
10   mylist.push_back (100);
11   mylist.push_back (200);
12   mylist.push_back (300);
13
14   cout << "mylist contains:";
15   for (it=mylist.begin(); it!=mylist.end(); ++it)
16     cout << " " << *it;
17
18   mylist.clear();
19   mylist.push_back (1101);
20   mylist.push_back (2202);
21
22   cout << "\nmylist contains:";
23   for (it=mylist.begin(); it!=mylist.end(); ++it)
24     cout << " " << *it;
25
26   cout << endl;
27
28   return 0;
29 }
30
```

1.42. list::push_front

```
void push_front ( const T& x );
```

Inserta un nuevo elemento al comienzo de list a la izquierda de su primer elemento. El contenido de este nuevo elemento es inicializado como una copia de x. Incrementa el tamaño de la lista en uno.

Ejemplo

```
   // list::push_front
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist (2,100);         // two ints with a value of 100
8    mylist.push_front (200);
9    mylist.push_front (300);
10
11   cout << "mylist contains:";
12   for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
13     cout << " " << *it;
14
15   cout << endl;
16   return 0;
17 }
18
```

1.43. list::pop_front

```
void pop_front ( );
```

Elimina el primer elemento de la lista, reduciendo en uno el actual tamaño. Se llama al destructor del elemento que se elimina.

Ejemplo

```
   // list::pop_front
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
8    mylist.push_back (100);
9    mylist.push_back (200);
10   mylist.push_back (300);
11
12   cout << "Popping out the elements in mylist:";
13   while (!mylist.empty())
14   {
15     cout << " " << mylist.front();
16     mylist.pop_front();
17   }
18
19   cout << "\nFinal size of mylist is " << int(mylist.size()) << endl;
20
21   return 0;
22 }
23
```

1.44. list::push_back

```
void push_back ( const T& x );
```

Añade un nuevo elemento al final de la lista, a la derecha de su último elemento. El contenido de este nuevo elemento se inicializa como una copia de x.
El tamaño de la lista se incrementa en uno

Ejemplo

```
   // list::push_back
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
8    int myint;
9
10   cout << "Please enter some integers (enter 0 to end):\n";
11
12   do {
13     cin >> myint;
14     mylist.push_back (myint);
15   } while (myint);
16
17   cout << "mylist stores " << (int) mylist.size() << " numbers.\n";
18
19   return 0;
20 }
21
```

1.45. list::pop_back

```
void pop_back ( );
```

Elimina el último elemento de la lista, reduciendo el tamaño de la lista en uno.
Se llama al destructor del elemento eliminado.

Ejemplo

```
   // list::pop_back
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
```

```
8     int sum (0);
9     mylist.push_back (100);
10    mylist.push_back (200);
11    mylist.push_back (300);
12
13    while (!mylist.empty())
14    {
15      sum+=mylist.back();
16      mylist.pop_back();
17    }
18
19    cout << "The elements of mylist summed " << sum << endl;
20
21    return 0;
22  }
23
```

1.46. list::splice

```
void splice ( iterator position, list<T,Allocator>& x );
void splice ( iterator position, list<T,Allocator>& x, iterator i );
void splice ( iterator position, list<T,Allocator>& x, iterator first, iterator
last );
```

Mueve los elementos de una lista a otra. Mueve los elementos de la lista x a la lista apuntada por this a partir de la position especificada por position. Los elementos que se mueven son eliminados de x. Se incrementa el tamaño de la lista y se reduce en la misma cantidad el tamaño de x. *this y x son listas diferentes.

Esta operación no implica construcción ni destrucción de elementos.

Ejemplo

```
1   // splicing lists
2   #include <iostream>
3   #include <list>
4   using namespace std;
5
6   int main ()
7   {
8     list<int> mylist1, mylist2;
9     list<int>::iterator it;
10
11    // set some initial values:
12    for (int i=1; i<=4; i++)
13       mylist1.push_back(i);      // mylist1: 1 2 3 4
14
15    for (int i=1; i<=3; i++)
16       mylist2.push_back(i*10);   // mylist2: 10 20 30
17
18    it = mylist1.begin();
19    ++it;                         // points to 2
20
21    mylist1.splice (it, mylist2); // mylist1: 1 10 20 30 2 3 4
22                                  // mylist2 (empty)
23                                  // "it" still points to 2 (the 5th element)
24
```

```
    mylist2.splice (mylist2.begin(),mylist1, it);
25                                  // mylist1: 1 10 20 30 3 4
26                                  // mylist2: 2
27                                  // "it" is now invalid.
28  it = mylist1.begin();
29  advance(it,3);                  // "it" points now to 30
30
31  mylist1.splice ( mylist1.begin(), mylist1, it, mylist1.end());
32                                  // mylist1: 30 3 4 1 10 20
33
34  cout << "mylist1 contains:";
35  for (it=mylist1.begin(); it!=mylist1.end(); it++)
36    cout << " " << *it;
37
38  cout << "\nmylist2 contains:";
39  for (it=mylist2.begin(); it!=mylist2.end(); it++)
40    cout << " " << *it;
41  cout << endl;
42
43  return 0;
44 }
45
```

1.47. list::remove

```
void remove ( const T& value );
```

Elimina elementos de la lista con el valor dado por value. Se invoca al destructor de esos elementos y se reduce el tamaño de la lista por el número de elementos que hubiera en la lista con dicho valor.

Ejemplo

```
   // remove from list
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    int myints[]= {17,89,7,14};
8    list<int> mylist (myints,myints+4);
9
10   mylist.remove(89);
11
12   cout << "mylist contains:";
13   for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
14     cout << " " << *it;
15   cout << endl;
16
17   return 0;
18 }
19
```

1.48. list ::remove_if

```
template <class Predicate>
  void remove_if ( Predicate pred );
```

Elimina elementos de la lista que cumplen la condición dada por pred. Esta llamada destruye estos objetos y reduce el tamaño de la lista por la cantidad de los elementos eliminados.

Ejemplo

```
   // list::remove_if
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  // a predicate implemented as a function:
6  bool single_digit (const int& value) { return (value<10); }
7
8  // a predicate implemented as a class:
9  class is_odd
10 {
11 public:
12   bool operator() (const int& value) {return (value%2)==1; }
13 };
14
15 int main ()
16 {
17   int myints[]= {15,36,7,17,20,39,4,1};
18   list<int> mylist (myints,myints+8);   // 15 36 7 17 20 39 4 1
19
20   mylist.remove_if (single_digit);      // 15 36 17 20 39
21
22   mylist.remove_if (is_odd());          // 36 20
23
24   cout << "mylist contains:";
25   for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
26     cout << " " << *it;
27   cout << endl;
28
29   return 0;
30 }
31
```

1.49. list::unique

```
  void unique ( );
template <class BinaryPredicate>
  void unique ( BinaryPredicate binary_pred );
```

Elimina valores duplicados. La primera versión, sin parámetros, deja solamente una ocurrencia por cada uno de los elemtnos repetidos que se presentan consecutivos en la lista. Es muy útil cuando tenemos listas ordenadas.
Para la segunda versión, aceptando un predicado binario y dando la función de comparación para determinar la unicidad de un elemento, la función llama a binary_pred(*i,*(i-1)) para todos los pares de elementos (siendo i un iterador) y elimina i de la lista si el predicado es

verdad.
Se llama a los destructores de los elementos eliminados y los iteradores y referencias a estos elementos se invalidan.
Ejemplo

```
   // list::unique
1  #include <iostream>
2  #include <cmath>
3  #include <list>
4  using namespace std;
5
6  // a binary predicate implemented as a function:
7  bool same_integral_part (double first, double second)
8  { return ( int(first)==int(second) ); }
9
10 // a binary predicate implemented as a class:
11 class is_near
12 {
13 public:
14   bool operator() (double first, double second)
15   { return (fabs(first-second)<5.0); }
16 };
17
18 int main ()
19 {
20   double mydoubles[]={ 12.15,  2.72, 73.0,  12.77,  3.14,
21                        12.77, 73.35, 72.25, 15.3,  72.25 };
22   list<double> mylist (mydoubles,mydoubles+10);
23
24   mylist.sort();             //  2.72,  3.14, 12.15, 12.77, 12.77,
25                              // 15.3,  72.25, 72.25, 73.0,  73.35
26
27   mylist.unique();           //  2.72,  3.14, 12.15, 12.77
28                              // 15.3,  72.25, 73.0,  73.35
29
30   mylist.unique (same_integral_part);  //  2.72,  3.14, 12.15
31                                        // 15.3,  72.25, 73.0
32
33   mylist.unique (is_near());           //  2.72, 12.15, 72.25
34
35   cout << "mylist contains:";
36   for (list<double>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
37     cout << " " << *it;
38   cout << endl;
39
40   return 0;
41 }
42
```

1.50. list::merge

```
  void merge ( list<T,Allocator>& x );
template <class Compare>
  void merge ( list<T,Allocator>& x, Compare comp );
```

Mezcla listas ordenadas. Mezcla *x* en la lista,insertando todos los elementos de x en la lista en la posición correcta para que la lista se mantenga ordenada. Vacía x e incrementa el

tamaño de la lista. La segunda versión tiene el mismo comportamiento pero toma una función como segundo parámetro para realizar la comparación con el objetivo de determinar los puntos de insercción.

La mezcla se realiza usando dos iteradores: uno para iterar sobre x y otro para mantener la insercción en la lista; Durante la iteración de x, si el elemento actual de x es menor que el elemento en el punto de insercción en la lista el elemento se elimina de x y se inserta en la lista, en otro caso se avanza el punto de insercción en la lista. Esta operación se repite hasta que se llega al final de la lista, en este caso todos los elementos que restan en x se mueven a la lista o bien se llega al final de x. No se construye ni se destruye ningún elemento.

Ejemplo

```
   // list::merge
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  // this compares equal two doubles if
6  //  their interger equivalents are equal
7  bool mycomparison (double first, double second)
8  { return ( int(first)<int(second) ); }
9
10 int main ()
11 {
12   list<double> first, second;
13
14   first.push_back (3.1);
15   first.push_back (2.2);
16   first.push_back (2.9);
17
18   second.push_back (3.7);
19   second.push_back (7.1);
20   second.push_back (1.4);
21
22   first.sort();
23   second.sort();
24
25   first.merge(second);
26
27   second.push_back (2.1);
28
29   first.merge(second,mycomparison);
30
31   cout << "first contains:";
32   for (list<double>::iterator it=first.begin(); it!=first.end(); ++it)
33     cout << " " << *it;
34   cout << endl;
35
36   return 0;
37 }
38
```

1.51. list::sort

```
  void sort ( );
template <class Compare>
  void sort ( Compare comp );
```

Ordena los elementos en el contenedor de menor a mayor.
En la primera versión, sin ningún parámetro, la comparación se realiza usando el operator< entre los elementos a ser comparados.
En la segunda versión, la comparación ser realiza usando la función comp.
La operación no realiza ninguna construcción o destrucción de ningún elemento.

Ejemplo

```
   // list::sort
1  #include <iostream>
2  #include <list>
3  #include <string>
4  #include <cctype>
5  using namespace std;
6
7  // comparison, not case sensitive.
8  bool compare_nocase (string first, string second)
9  {
10   unsigned int i=0;
11   while ( (i<first.length()) && (i<second.length()) )
12   {
13     if (tolower(first[i])<tolower(second[i])) return true;
14     else if (tolower(first[i])>tolower(second[i])) return false;
15     ++i;
16   }
17   if (first.length()<second.length()) return true;
18   else return false;
19 }
20
21 int main ()
22 {
23   list<string> mylist;
24   list<string>::iterator it;
25   mylist.push_back ("one");
26   mylist.push_back ("two");
27   mylist.push_back ("Three");
28
29   mylist.sort();
30
31   cout << "mylist contains:";
32   for (it=mylist.begin(); it!=mylist.end(); ++it)
33     cout << " " << *it;
34   cout << endl;
35
36   mylist.sort(compare_nocase);
37
38   cout << "mylist contains:";
39   for (it=mylist.begin(); it!=mylist.end(); ++it)
40     cout << " " << *it;
41   cout << endl;
42
43   return 0;
44 }
45
```

1.52. list::reverse

```
void reverse ( );
```

Invierte el orden de los elementos en la lista. Todos los iteradores y referencias a los elementos permanecen válidos.

Ejemplo

```
   // reversing vector
1  #include <iostream>
2  #include <list>
3  using namespace std;
4
5  int main ()
6  {
7    list<int> mylist;
8    list<int>::iterator it;
9
10   for (int i=1; i<10; i++) mylist.push_back(i);
11
12   mylist.reverse();
13
14   cout << "mylist contains:";
15   for (it=mylist.begin(); it!=mylist.end(); ++it)
16     cout << " " << *it;
17
18   cout << endl;
19
20   return 0;
21 }
22
```

Pilas (Stack)

class template

<stack>

1.53. stack::stack

```
explicit stack ( const Container& ctnr = Container() );
```

Construct stack

Construye un stack. Es posible construir una pila vacía declarando el contenedor sin ningún parámetro. Pero tambien podemos construirlos conteniendo una copia de los elementos existentes en otro contenedor (vector, lista, cola, etc).

Ejemplo

```
1  // constructing stacks
2  #include <iostream>
3  #include <vector>
4  #include <deque>
5  #include <stack>
6  using namespace std;
```

```
7  int main ()
8  {
9    list<int> mylist (3,100);     // list with 3 elements
10   vector<int> myvector (2,200);   // vector with 2 elements
11
12   stack<int> first;               // empty stack
13   stack<int> second (list);    // stack initialized to copy of mylist
14
15   stack<int,vector<int> > third;  // empty stack using vector
16   stack<int,vector<int> > fourth (myvector);
17
18   cout << "size of first: " << (int) first.size() << endl;
19   cout << "size of second: " << (int) second.size() << endl;
20   cout << "size of third: " << (int) third.size() << endl;
21   cout << "size of fourth: " << (int) fourth.size() << endl;
22
23   return 0;
24 }
25
```

1.54. stack::size

```
size_type size ( ) const;
```

Devuelve el número de elementos de la pila (stack).

El tipo size_type es un entero sin signo.

Ejemplo

```
   // stack::size
1  #include <iostream>
2  #include <stack>
3  using namespace std;
4
5  int main ()
6  {
7    stack<int> myints;
8    cout << "0. size: " << (int) myints.size() << endl;
9
10   for (int i=0; i<5; i++) myints.push(i);
11   cout << "1. size: " << (int) myints.size() << endl;
12
13   myints.pop();
14   cout << "2. size: " << (int) myints.size() << endl;
15
16   return 0;
17 }
18
```

1.55. stack::empty

```
bool empty ( ) const;
```

Devuelve si la pila esta vacía, es decir, si su tamaño es 0.
Ejemplo

```
   // stack::empty
1  #include <iostream>
2  #include <stack>
3  using namespace std;
4
5  int main ()
6  {
7    stack<int> mystack;
8    int sum (0);
9
10   for (int i=1;i<=10;i++) mystack.push(i);
11
12   while (!mystack.empty())
13   {
14      sum += mystack.top();
15      mystack.pop();
16   }
17
18   cout << "total: " << sum << endl;
19
20   return 0;
21 }
22
```

1.56. stack::top

```
      value_type& top ( );
const value_type& top ( ) const;
```

Devuelve una referencia al siguiente elemento de la pila. Ya que las pilas tienen la política del último en entrar es el primero en salir, este elemento que se devuelve también fue el último en insertarse.

Ejemplo

```
   // stack::top
1  #include <iostream>
2  #include <stack>
3  using namespace std;
4
5  int main ()
6  {
7    stack<int> mystack;
8
9    mystack.push(10);
10   mystack.push(20);
11
12   mystack.top() -= 5;
13
14   cout << "mystack.top() is now " << mystack.top() << endl;
15
16   return 0;
17
18
```

```
}
```

1.57. stack::push

```
void push ( const T& x );
```

Añade un nuevo elemento al tope (“top”) de la pila, el contenido de este nuevo elemento es una copia de x.

Ejemplo

```
   // stack::push/pop
1  #include <iostream>
2  #include <stack>
3  using namespace std;
4
5  int main ()
6  {
7    stack<int> mystack;
8
9    for (int i=0; i<5; ++i) mystack.push(i);
10
11   cout << "Popping out elements...";
12   while (!mystack.empty())
13   {
14      cout << " " << mystack.top();
15      mystack.pop();
16   }
17   cout << endl;
18
19   return 0;
20 }
21
```

1.58. stack::pop

```
void pop ( );
```

Elimina el elemento en la posición tope de la pila, reduciendo el tamaño de la pila en uno. El valor de este elemento puede ser salvado si previamente llamamos a la función top. Se llama al destructor del elemento eliminado.

Ejemplo

```
1  // stack::push/pop
2  #include <iostream>
3  #include <stack>
4  using namespace std;
5
6  int main ()
7  {
8    stack<int> mystack;
9
```

```
    for (int i=0; i<5; ++i) mystack.push(i);

    cout << "Popping out elements...";
    while (!mystack.empty())
    {
       cout << " " << mystack.top();
       mystack.pop();
    }
    cout << endl;

    return 0;
}
```

Colas (Queue)

class template
<queue>

Tiene las mismas funcionalidades que **STACK** más las dos siguientes:

1.59. queue::front

```
      value_type& front ( );
const value_type& front ( ) const;
```

Devuelve una referencia al siguiente elemento de la cola. Este es elemento más antiguo de la cola siendo este el que se borraría si se invoca a la función pop.

Ejemplo

```
   // queue::front
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
  queue<int> myqueue;

  myqueue.push(77);
  myqueue.push(16);

  myqueue.front() -= myqueue.back();    // 77-16=61

  cout << "myqueue.front() is now " << myqueue.front() << endl;

  return 0;
}
```

1.60. queue::back

```
      value_type& back ( );
const value_type& back ( ) const;
```

Devuelve una referencia al último elemento de la cola. Este esl elemento más reciente que fue insertado o el último elemento puesto en la cola.

Ejemplo

```
   // queue::front
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4
5  int main ()
6  {
7    queue<int> myqueue;
8
9    myqueue.push(12);
10   myqueue.push(75);   // this is now the back
11
12   myqueue.back() -= myqueue.front();
13
14   cout << "myqueue.back() is now " << myqueue.back() << endl;
15
16   return 0;
17 }
18
```

Colas con Prioridad (Priority_Queue)

class template

<queue>

Tiene las mismas funcionalidades que Queue, exceptuando las funciones miembros **front** y **back,** y añadiendo las función top.

1.61. prority_queue::top

```
const value_type& top ( ) const;
```

Devuelve una referencia constante al elemento tope de la cola con prioridad. El elemento situado en el tope es aquel que tiene la mayor prioridad. Este elemento es el que se borra cuando se invoca a la función pop.

Ejemplo

```
   // priority_queue::top
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4
```

```
5   int main ()
6   {
7     priority_queue<int> mypq;
8
9     mypq.push(10);
10    mypq.push(20);
11    mypq.push(15);
12
13    cout << "mypq.top() is now " << mypq.top() << endl;
14
15    return 0;
16  }
17
```

Conjunto (Set)

class template
<set>

Conjunto es un contenedor asociativo que almacena elementos unívocos, así estos elementos pueden actuar como llaves o claves (p.e de un fichero en disco).
En los contenedoes asociativos están diseñados para eficientemente acceder por su clave.

Internamente, los elementos en un conjunto están siempre ordenados de mayor a menor. Normalmente los conjunto se implementan como arboles binarios de búsqueda.

unordered_set, are available in implementations following TR1.

Este contenedor soporta iteradores bidireccionales.

El contenedor set usa tres parámetros plantilla:

```
   template < class Key, class Compare = less<Key>,
1             class Allocator = allocator<Key> > class set;
2
```

Teniendo cada uno de ellos el siguiente significado:

- **Key:** tipo Key: tipo de los elementos en el contendor. Cada elemento en un set es también su llave.
- **Compare:** Clase Comparision: Una clase que toma dos argumento del mismo tipo como los elementos del contenedor ydevuelve un bool. La expresión comp(a,b), siendo *comp* un objeto de este clase comparision y a y b dos elementos del contenedor, devolverá true si a está situado en una position previa a b. El valor por defecto es less<Key> que devuelve los mismo que si aplicamos el operador menor que (a<b). El objeto set object usa esta expresión para determinar la posición del elemento en el contendor. Todos los elementos están ordenados siguiendo esta regla.
- **Allocator:** Tipo de objeto asignador de memoria .Por defecto, la clase plantilla allocator para el tipo Key es la que se usa.

1.62. set::set

```
explicit set ( const Compare& comp = Compare(),
               const Allocator& = Allocator() );
template <class InputIterator>
  set ( InputIterator first, InputIterator last,
```

```
        const Compare& comp = Compare(), const Allocator& = Allocator() );
set ( const set<Key,Compare,Allocator>& x );
```

Construye un objeto conjunto, inicializando su contenido dependiendo sobre la versión del constructor que se use:

- explicit set (const Compare& comp = Compare(), Allocator& = Allocator());

  Constructor por defecto: constriye un objeto conjunto vacío, con ningún contenido y tamaño cero.

- template <class InputIterator> set (InputIterator first, InputIterator last, const Compare& comp= Compare(), const Allocator& = Allocator());

  Constructor por iteración:Itera entre first and last, poniendo una copia de los elementos en la secuencia como el contenido del objecto set.

- set (const set<Key,Compare,Allocator>& x);

  Constructor de Cpòa: El objeto es inicializado para tener el mismo contenido y propiedades como el objecto set x.

Ejemplo

```
   // constructing sets
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  bool fncomp (int lhs, int rhs) {return lhs<rhs;}
6
7  struct classcomp {
8    bool operator() (const int& lhs, const int& rhs) const
9    {return lhs<rhs;}
10 };
11
12 int main ()
13 {
14   set<int> first;                           // empty set of ints
15
16   int myints[]= {10,20,30,40,50};
17   set<int> second (myints,myints+5);        // pointers used as iterators
18
19   set<int> third (second);                  // a copy of second
20
21   set<int> fourth (second.begin(), second.end());  // iterator ctor.
22
23   set<int,classcomp> fifth;                 // class as Compare
24
25   bool(*fn_pt)(int,int) = fncomp;
26   set<int,bool(*)(int,int)> sixth (fn_pt);  // function pointer as Compare
27
28   return 0;
29 }
30
```

1.63. set::~set

```
~set ( );
```

Destruye el objeto contenedor. Este llama al destructor para cada uno de los elementos del conjunto y libera toda la memoria asignada al conjunto.

1.64. set::operator=

```
set<Key,Compare,Allocator>&
     operator= ( const set<Key,Compare,Allocator>& x );
```

Asigna una copia de los elementos en x como el nuevo contenido del conjunto. Los elementos que hubiesen el el conjunto previo a la llamada son eliminados, y sustituidos por copias de aquellos en el conjunto x.
Después de la llamada a esta función, ambos conjuntos tendrán el mismo tamaño y comparando elemento a elemento son iguales.
Ejemplo:

```
   // assignment operator with sets
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    int myints[]={ 12,82,37,64,15 };
8    set<int> first (myints,myints+5);   // set with 5 ints
9    set<int> second;                    // empty set
10
11   second=first;                       // now second contains the 5 ints
12   first=set<int>();                   // and first is empty
13
14   cout << "Size of first: " << int (first.size()) << endl;
15   cout << "Size of second: " << int (second.size()) << endl;
16   return 0;
17 }
18
```

1.65. set::begin

```
      iterator begin ();
const_iterator begin () const;
```

Devuelve un iterador apuntando al primer elmento en el contenedor set. Internamente, el contenedor set mantiene sus elementos ordenados de menor a mayor, por lo tanto
`begin` devuelve el elemento con menor valor de la llave.

Ejemplo

```
1  // set::begin/end
```

```
   #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    int myints[] = {75,23,65,42,13};
8    set<int> myset (myints,myints+5);
9
10   set<int>::iterator it;
11
12   cout << "myset contains:";
13   for ( it=myset.begin() ; it != myset.end(); it++ )
14     cout << " " << *it;
15
16   cout << endl;
17
18   return 0;
19 }
20
```

1.66. set::end

```
      iterator end ();
const_iterator end () const;
```

Devuelve un iterador al elemento siguiente al último elemento del contenedor.

Ambos `iterator` and `const_iterator` son tipos miembros. En la clase plantilla set estos iteradores son bidireccionales.

Ejemplo

```
   // set::begin/end
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    int myints[] = {75,23,65,42,13};
8    set<int> myset (myints,myints+5);
9
10   set<int>::iterator it;
11
12   cout << "myset contains:";
13   for ( it=myset.begin() ; it != myset.end(); it++ )
14     cout << " " << *it;
15
16   cout << endl;
17
18   return 0;
19 }
20
```

1.67. set::rbegin

```
      reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

Devuelve un iterador inverso apuntando al último elemento del contendor set. Internamente, el contendor set mantiene sus elementos ordenados de menor a mayor, por lo tanto `rbegin` devuelve el elemento con la mayor clave

Note que `rbegin` no se refiere al mismo elemento que indica end, pero si al elemento a la derecha de él.

Ejemplo

```
   // set::rbegin/rend
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    int myints[] = {78,21,64,49,17};
8    set<int> myset (myints,myints+5);
9
10   set<int>::reverse_iterator rit;
11
12   cout << "myset contains:";
13   for ( rit=myset.rbegin() ; rit != myset.rend(); rit++ )
14     cout << " " << *rit;
15
16   cout << endl;
17
18   return 0;
19 }
20
```

1.68. set::rend

```
      reverse_iterator rend();
const_reverse_iterator rend() const;
```

Devuelve un iterador inverso apuntando al elemento a la izquierda del primer elemento, que es considereado el final inverso.

Notice that `rend` does not refer to the same element as begin, but to the element right before it.

Ejemplo

```
   // set::rbegin/rend
1  #include <iostream>
2  #include <set>
3
```

```
   using namespace std;
4
5  int main ()
6  {
7    int myints[] = {78,21,64,49,17};
8    set<int> myset (myints,myints+5);
9
10   set<int>::reverse_iterator rit;
11
12   cout << "myset contains:";
13   for ( rit=myset.rbegin() ; rit != myset.rend(); rit++ )
14     cout << " " << *rit;
15
16   cout << endl;
17
18   return 0;
19 }
20
```

1.69. set::size

```
size_type size() const;
```

Devuelve el numero de elementos del contenedor.

Ejemplo

```
   // set::size
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    set<int> myints;
8    cout << "0. size: " << (int) myints.size() << endl;
9
10   for (int i=0; i<10; i++) myints.insert(i);
11   cout << "1. size: " << (int) myints.size() << endl;
12
13   myints.insert (100);
14   cout << "2. size: " << (int) myints.size() << endl;
15
16   myints.erase(5);
17   cout << "3. size: " << (int) myints.size() << endl;
18
19   return 0;
20 }
21
```

1.70. set::max_size

```
size_type max_size () const;
```

Devuelve el número máximo de elementos que el contendor conjunto puede mantener. Este es el tamaño potencial máximo que el contendor puede alcanzar debido al sistema o a las limitaciones de biblioteca usada.

Ejemplo

```
   // set::max_size
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    int i;
8    set<int> myset;
9
10   if (myset.max_size()>1000)
11   {
12     for (i=0; i<1000; i++) myset.insert(i);
13     cout << "The set contains 1000 elements.\n";
14   }
15   else cout << "The set could not hold 1000 elements.\n";
16
17   return 0;
18 }
19
```

1.71. set::empty

```
bool empty ( ) const;
```

Devuelve si el conjunto es vacío, p.e si el tamaño es 0.
Esta función no modifica el contenido del contenedor. Para eliminar su contenido se usa la función clear.

Ejemplo

```
   // set::empty
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    set<int> myset;
8
9
10   myset.insert(20);
11   myset.insert(30);
12   myset.insert(10);
13
14   cout << "myset contains:";
15   while (!myset.empty())
16   {
17      cout << " " << *myset.begin();
```

```
      myset.erase(myset.begin());
18   }
19   cout << endl;
20
21   return 0;
22 }
23
```

1.72. set::insert

```
pair<iterator,bool> insert ( const value_type& x );
          iterator insert ( iterator position, const value_type& x );
template <class InputIterator>
     void insert ( InputIterator first, InputIterator last );
```

Insertar un elemento. Al conjunto se le inserta nuevos elementos (uno si el parámetro x se usa) o se le inserta la secuencia de elementos (si los iteradores de entrada son los parámetros de entrada). Esta operación va a incrementar el tamaño del conjunto por la cantidad de elementos insertados.
Ya que el conjunto no permite valores duplicado, la operación de insercción comprueba para cada elemento a ser insertado si existe ya en el contendor otro elemento con el mismo valor, si es así, el elemento no se inserta.

La primera versión devuelve un objeto de tipo pair, cuyo primer miembro `pair::first` es un iterador apuntando o bien al nuevo elemento insertado o al elemento que existía ya en el conjunto con el mismo valor. El elemento `pair::second` es un booleano con valor true si el elemento fue insertado o false en caso contrario.

Ejemplo

```
1   // set::insert
2   #include <iostream>
3   #include <set>
4   using namespace std;
5
6   int main ()
7   {
8     set<int> myset;
9     set<int>::iterator it;
10    pair<set<int>::iterator,bool> ret;
11
12    // set some initial values:
13    for (int i=1; i<=5; i++) myset.insert(i*10);    // set: 10 20 30 40 50
14
15    ret = myset.insert(20);               // no new element inserted
16
17    if (ret.second==false) it=ret.first;  // "it" now points to element 20
18
19    myset.insert (it,25);                 // max efficiency inserting
20    myset.insert (it,24);                 // max efficiency inserting
21    myset.insert (it,26);                 // no max efficiency inserting
22
23    int myints[]= {5,10,15};              // 10 already in set, not inserted
24    myset.insert (myints,myints+3);
25
26    cout << "myset contains:";
      for (it=myset.begin(); it!=myset.end(); it++)
```

```
27      cout << " " << *it;
28    cout << endl;
29
30    return 0;
31  }
32
```

1.73. set::erase

```
     void erase ( iterator position );
size_type erase ( const key_type& x );
     void erase ( iterator first, iterator last );
```

Elimina elementos del conjunto con la posibilidad de eliminar un único elemento dado por la posición position o elimina varios elementos. Se puede eliminar también todos aquellos elementos con clave x o todos aquellos en el rango que marca los iteradores (`[first,last)`). Se disminuye el tamaño del contendedor por el número de elementos eliminados.
Ejemplo

```
    // erasing from set
1   #include <iostream>
2   #include <set>
3   using namespace std;
4
5   int main ()
6   {
7     set<int> myset;
8     set<int>::iterator it;
9
10    // insert some values:
11    for (int i=1; i<10; i++) myset.insert(i*10);  // 10 20 30 40 50 60 70 80 90
12
13    it=myset.begin();
14    it++;                                         // "it" points now to 20
15
16    myset.erase (it);
17
18    myset.erase (40);
19
20    it=myset.find (60);
21    myset.erase ( it, myset.end() );
22
23    cout << "myset contains:";
24    for (it=myset.begin(); it!=myset.end(); ++it)
25      cout << " " << *it;
26    cout << endl;
27
28    return 0;
29  }
30
```

1.74. set::swap

```
void swap ( set<Key,Compare,Allocator>& st );
```

Intercambia los contenidos del contenedor con el contenido de set, que es otro conjunto con elementos del mismo tipo. Los tamaños de los conjuntos pueden ser diferentes.
Ejemplo

```
1  // swap sets
2  #include <iostream>
3  #include <set>
4  using namespace std;
5
6  main ()
7  {
8    int myints[]={12,75,10,32,20,25};
9    set<int> first (myints,myints+3);     // 10,12,75
10   set<int> second (myints+3,myints+6);  // 20,25,32
11   set<int>::iterator it;
12
13   first.swap(second);
14
15   cout << "first contains:";
16   for (it=first.begin(); it!=first.end(); it++) cout << " " << *it;
17
18   cout << "\nsecond contains:";
19   for (it=second.begin(); it!=second.end(); it++) cout << " " << *it;
20
21   cout << endl;
22
23   return 0;
24 }
```

1.75. set:clear

```
void clear ( );
```

Todos los elementos del conjunto son eliminados: se llama a sus destructores, y son eliminados a continuación del contenedor, dejándolo con tamaño 0.

Ejemplo

```
1  // set::clear
2  #include <iostream>
3  #include <set>
4  using namespace std;
5
6  int main ()
7  {
8    set<int> myset;
9    set<int>::iterator it;
10
11   myset.insert (100);
12   myset.insert (200);
13   myset.insert (300);
14
15   cout << "myset contains:";
16   for (it=myset.begin(); it!=myset.end(); ++it)
17     cout << " " << *it;
```

```
18    myset.clear();
19    myset.insert (1101);
20    myset.insert (2202);
21
22    cout << "\nmyset contains:";
23    for (it=myset.begin(); it!=myset.end(); ++it)
24      cout << " " << *it;
25
26    cout << endl;
27
28    return 0;
29  }
30
```

1.76. set::key_comp

```
key_compare key_comp ( ) const;
```

Devuelve un objeto comparación asociado al contenedor, que puede ser usado para comparar dos objetos.
Este objeto comparación puede bien ser un puntero a una función o un objeto de una clase con un operador. En ambos caso toma dos argumentos del mismo tipo que el tipo de los elementos del contendor, y devuelve true si el primero argumento se considera a ir previo que el segundo o false en caso contrario.

`set::key_compare` es un tipo miembro definido para `Compare`, que es el segundo parámetro plantilla definido en la clase plantilla set.

Ejemplo

```
1   // set::key_comp
2   #include <iostream>
3   #include <set>
4   using namespace std;
5
6   int main ()
7   {
8     set<int> myset;
9     set<int>::key_compare mycomp;
10    set<int>::iterator it;
11    int i,highest;
12
13    mycomp = myset.key_comp();
14
15    for (i=0; i<=5; i++) myset.insert(i);
16
17    cout << "myset contains:";
18
19    highest=*myset.rbegin();
20    it=myset.begin();
21    do {
22      cout << " " << *it;
23    } while ( mycomp(*it++,highest) );
24
      cout << endl;
```

```
25
26    return 0;
27  }
```

1.77. set::value_comp

```
value_compare value_comp ( ) const;
```

Devuelve un objeto comparación asociado al contenedor, que puede ser usado para comparar dos objetos.
Este objeto comparación puede bien ser un puntero a una función o un objeto de una clase con un operador. En ambos caso toma dos argumentos del mismo tipo que el tipo de los elementos del contendor, y devuelve true si el primero argumento se considera a ir previo que el segundo o false en caso contrario.

`set::value_compare` es un tipo miembro definido para `Compare`,que es el segundo parámetro plantilla definido en la clase plantilla set.

Ejemplo

```
    // set::value_comp
1   #include <iostream>
2   #include <set>
3   using namespace std;
4
5   int main ()
6   {
7     set<int> myset;
8     set<int>::value_compare mycomp;
9     set<int>::iterator it;
10    int i,highest;
11
12    mycomp = myset.value_comp();
13
14    for (i=0; i<=5; i++) myset.insert(i);
15
16    cout << "myset contains:";
17
18    highest=*myset.rbegin();
19    it=myset.begin();
20    do {
21      cout << " " << *it;
22    } while ( mycomp(*it++,highest) );
23
24    cout << endl;
25
26    return 0;
27  }
```

1.78. set::find

```
iterator find ( const key_type& x ) const;
```

Devuelve un iterador a un elemento, con clave x, del conjunto. Si no lo encuentra devuelve un iterador a set::end.

Ejemplo

```
   // set::find
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    set<int> myset;
8    set<int>::iterator it;
9
10   // set some initial values:
11   for (int i=1; i<=5; i++) myset.insert(i*10);    // set: 10 20 30 40 50
12
13   it=myset.find(20);
14   myset.erase (it);
15   myset.erase (myset.find(40));
16
17   cout << "myset contains:";
18   for (it=myset.begin(); it!=myset.end(); it++)
19     cout << " " << *it;
20   cout << endl;
21
22   return 0;
23 }
24
```

1.79. set::count

```
size_type count ( const key_type& x ) const;
```

Busca en el contenedor los elementos con llave x y devuelve el número de veces que aparece en el contendor. Ya que en un conjunto no puede existir elementos repetidos, esto significa que la función devuelve, en el caso de que exista la clave x en el conjunto, 1 y en otro caso 0.

Ejemplo

```
   // set::count
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    set<int> myset;
8    int i;
9
10   // set some initial values:
11
```

```
12
13    for (i=1; i<5; i++) myset.insert(i*3);    // set: 3 6 9 12
14
15    for (i=0;i<10; i++)
16    {
17      cout << i;
18      if (myset.count(i)>0)
19        cout << " is an element of myset.\n";
20      else
21        cout << " is not an element of myset.\n";
22    }
23
24
```

1.80. set::lower_bound

```
iterator lower_bound ( const key_type& x ) const;
```

Devuelve un iterador que apunta el primer elemento del contenedor siendo mayor o igual que x.
Note que, internamente, todos los elementos en un conjunto son siempre ordenados siguiendo el criterio definido por sus objeto comparision, por lo tanto todos los elementos que están a continuación del iterador que la función devuelve son mayores o iguales a x.

Ejemplo

```
    // set::lower_bound/upper_bound
1   #include <iostream>
2   #include <set>
3   using namespace std;
4
5   int main ()
6   {
7     set<int> myset;
8     set<int>::iterator it,itlow,itup;
9
10    for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90
11
12    itlow=myset.lower_bound (30);                //       ^
13    itup=myset.upper_bound (60);                 //                     ^
14
15    myset.erase(itlow,itup);                     // 10 20 70 80 90
16
17    cout << "myset contains:";
18    for (it=myset.begin(); it!=myset.end(); it++)
19      cout << " " << *it;
20    cout << endl;
21
22    return 0;
23  }
24
```

1.81. set::upper_bound

```
iterator upper_bound ( const key_type& x ) const;
```

Devuelve un iterador apuntando al primer elemento en el contenedor que es estrictamente mayor que x.

A diferencia de lower_bound, esta función miembro no devuelve un iterador al elemento que sea igual a x, pero sí al estrictamente mayor.

Ejemplo

```
   // set::lower_bound/upper_bound
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    set<int> myset;
8    set<int>::iterator it,itlow,itup;
9
10   for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90
11
12   itlow=myset.lower_bound (30);                //       ^
13   itup=myset.upper_bound (60);                 //                   ^
14
15   myset.erase(itlow,itup);                     // 10 20 70 80 90
16
17   cout << "myset contains:";
18   for (it=myset.begin(); it!=myset.end(); it++)
19     cout << " " << *it;
20   cout << endl;
21
22   return 0;
23 }
24
```

1.82. set::equal_range

```
pair<iterator,iterator> equal_range ( const key_type& x ) const;
```

Devuelve el rango de elementos que son iguales a x. Devuelve los límites de un rango que incluye todos los elementos que tiene como clave x. En la clase conjunto al no haber claves repetidas el rango como máximo incluye un elemento.
Si x no aparece en el conjunto, el rango tiene longitud ceor, con ambos iteradores apuntando al valor más próximo mayor que x, si existe, o en caso contrario apuntando a set:end si x es mayor que todos los elementos en el conenedor.
Ejemplo

```
   // set::equal_elements
1  #include <iostream>
2
```

```
   #include <set>
3  using namespace std;
4
5  int main ()
6  {
7    set<int> myset;
8    pair<set<int>::iterator,set<int>::iterator> ret;
9
10   for (int i=1; i<=5; i++) myset.insert(i*10);   // set: 10 20 30 40 50
11
12   ret = myset.equal_range(30);
13
14   cout << "lower bound points to: " << *ret.first << endl;
15   cout << "upper bound points to: " << *ret.second << endl;
16
17   return 0;
18 }
19
```

multiset

class template
<set>

Conjunto con múltiples claves

Los contendores `multiset` tienen las mismas propiedades que los contendores set, pero permitiendo claves múltiples con valores iguales.
En su implementación en la C++ STL, los contenedores `multiset` tienen los mismos tres parámetros plantilla como el contenedor set:

```
  template < class Key, class Compare = less<Key>,
1            class Allocator = allocator<Key> > class multiset;
2
```

Teniendo cada uno de ellos el siguiente significado:

- **Key:** tipo Key: tipo de los elementos en el contendor.Cada elemento en un `set` es también su llave.
- **Compare:** Clase Comparision: Una clase que toma dos argumento del mismo tipo como los elementos del contenedor ydevuelve un bool. La expresión `comp(a,b)`, siendo *comp* un objeto de este clase comparision y a y b dos elementos del contenedor, devolverá true si a está situado en una position previa a b. El valor por defecto es `less<Key>` que devuelve los mismo que si aplicamos el operador menor que (a<b). El objeto `set` object usa esta expresión para determinar la posición del elemento en el contendor. Todos los elementos están ordenados siguiendo esta regla.
- **Allocator:** Tipo de objeto asignador de memoria .Por defecto, la clase plantilla `allocator` para el tipo `Key` es la que se usa.

La clase plantilla `multiset` esta definida en el fichero cabecera `<set>`.

Para mayor detalle de la funcionalidad consultar la clase set.

map

class template
<map>

Map

Los contendores Map (diccionarios) son un tipo de contenedor asociativo que almacena elementos formados por la combinación de clave y valor asociado (o "mapped").

En un `map`, el valor clave se usa para identificar el elemento, mientras que el valor asociado está relacionado con el valor clave. Por ejemplo un ejemplo común de un contendor map es una guía de teléfonos en la que el nombre es el valor clave y el número de telefono el el valor asociado.

Internamente, los elementos en un map son ordenados de menor a mayor por el valor de la clave.
Los contendores asociativos, están diseñados para realizar las consultas de forma eficiente por su valor clave (a diferencia de los contenedores secuenciales, que son más eficientes acceder por su relativa o absoluta posición).

Por lo tanto, las principales características de una contenedor `map` son:

- Unicidad en los valores clave: en un contendor `map` no puede haber claves repetidas.
- Cada elemento se compone de clave y valor asociado
- Los elementos están ordenados por su valor clave.

Los contenedores Maps son los únicos contenedores asociativos que implementan el operador acceso directo (`operator[]`).
En la implementacion STL C++, los contenedores `map` toman cuatro parámetros plantilla:

```
  template < class Key, class T, class Compare = less<Key>,
1            class Allocator = allocator<pair<const Key,T> > > class map;
2
```

El significado de cadauno de estos parámetros son:

- **Key:** Tipo de los valores clave.Cada elemento en un `map` es unívocamente identificado por su valor clave.
- **T:** Tipo del valor asociado. Cada elemento en un `map` se usa para almacenar un valor asociado correspondiente a un valor clave.
- **Compare:** Clase comparación: Una clase que toma dos argumentos del tipo *key* y devuelve un `bool`. La expresión `comp(a,b)`, siendo un objeto de esta clase comparación y *a* y *b* son valores clave, devuelve `true` si a se sitúa antes que *b*. Por defecto `less<Key>`, devuelve los mismo que se aplicamos el operador < (`a<b`). El objeto `map` usa esta expresión para determinar la posición de los elementos en el contendor. Todos los elementos en un contendor `map` están ordernaod siguiendo las mismas reglas.
- **`Allocator`:** Tipo del objeto asignador definido para asignar memoria.

Este contendor soporta iteradores bidireccionales. Para poder acceder, a través de los iteradores del contenedor `map`,tanto a los valores key o valor asociado, la clase define lo que llama `value_type`, que es un objeto de la clase pair en el que su primer elemento corresponde a un tipo conts key y su segundo elemento corresponde a tipo T.

```
  typedef pair<const Key, T> value_type;
```

Iteradores de un contendor `map` apunta a elementos de `value_type`.Por lo tanto, para un iterador llamado `it`, que apunta a un elemento de un `map`, su clave y valor asociado puede ser dereferenciado respectivamente con:

```
  map<Key,T>::iterator it;
1 (*it).first;              // the key value (of type Key)
2 (*it).second;             // the mapped value (of type T)
3 (*it);                    // the "element value" (of type pair<const Key,T>)
4
```

Naturalmente, alternativamente pordemos usar `->` o `[]`

```
  it->first;              // same as (*it).first   (the key value)
1 it->second;             // same as (*it).second  (the mapped value)
2
```

Como las operaciones son las mismas que en set exceptuando el operador[] solamente comentaremos este último.

1.83. map::operator[]

```
T& operator[] ( const key_type& x );
```

Accede a un elemento. Si x coincide con la clave de algún elmento en el contendor, la función devuelve una referencia a su valor asociado.
Si x no coinciden con el valor clave de ningún elemento en el contendor, la función inserta un nuevo elemento con esa llave y devuelve una referecia a su valor asociado. Note que siempre se va a incrementar el tamaño por uno,incluso si ningún valor asociado se asigna al elemento (el elemento se construye usando su constructor por defecto),

Una llamada a esta función es equivalente a realizar: `(*((this->insert(make_pair(x,T()))).first)).second`

Ejemplo

```
   // accessing mapped values
1  #include <iostream>
2  #include <map>
3  #include <string>
4  using namespace std;
5
6  int main ()
7  {
8    map<char,string> mymap;
9
10   mymap['a']="an element";
11   mymap['b']="another element";
12   mymap['c']=mymap['b'];
13
14   cout << "mymap['a'] is " << mymap['a'] << endl;
15   cout << "mymap['b'] is " << mymap['b'] << endl;
16   cout << "mymap['c'] is " << mymap['c'] << endl;
17   cout << "mymap['d'] is " << mymap['d'] << endl;
18
19   cout << "mymap now contains " << (int) mymap.size() << " elements." << endl;
20
21   return 0;
22 }
```

1.84. multimap

class template
<map>

Map con multiples-claves (diccionarios con múltiples claves)

Los contenedores Multimaps son contenedores asociativos que almacenan elementos formados por una combinación valor clave y valor asociado, pero a diferencia de los contenedores map, estos pueden tener diferentes elementos con la misma clave.

En su implementación en la STL de C++, `multimap` contiene cuatro parámetros plantilla:

```
  template < class Key, class T, class Compare = less<Key>,
1            class Allocator = allocator<pair<const Key,T> > > class multimap;
2
```

Cada uno de esos parámetros tiene el siguiente significado:

- **Key:** Tipo de los valores clave.Cada elemento en un `multimap` es generalmente unívocamente identificado por su valor clave, aunque diferentes elementos pueden tener la misma clave.
- **T:** Tipo del valor asociado. Cada elemento en un `multimap` se usa para almacenar un valor asociado correspondiente a un valor clave.
- **Compare:** Clase comparación: Una clase que toma dos argumentos del tipo *key* y devuelve un `bool`. La expresión `comp(a,b)`, siendo un objeto de esta clase comparación y *a* y *b* son valores clave, devuelve `true` si a se sitúa antes que *b*. Por defecto `less<Key>`, devuelve los mismo que se aplicamos el operador `<` (`a<b`). El objeto `map` usa esta expresión para determinar la posición de los elementos en el contendor. Todos los elementos en un contendor `map` están ordernaod siguiendo las mismas reglas.
- **Allocator:** Tipo del objeto asignador definido para asignar memoria.

La clase plantilla `multimap` está definida en la cabecera `map`.

Las funciones asociadas a multimap son las mismas de map exceptuando que el operador[] no está definido en esta.

Bibliografía

- C++ Reference: STL Containers. http://www.cplusplus.com/reference/stl/
- STL Tutorial and Reference Guide. Alexander Stepanov. Addison-Wesley.