Programación con la STL: contenedores genéricos (II)

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Noviembre 2009

Outline

Outline

Adaptadores

Los adaptadores encapsulan contenedores de tipo secuencias en otros contenedores con una interfaz restringida, generalmente mas adaptada a una expresión de un problema dado, son típicas ADTs

```
template<typename _Tp,
         typename _Sequence = deque<_Tp> >
    class someAdaptor;
```

Contenedor stack

- push()
- pop()
- top()

Contenedor stack

- push()
- pop()
- top()

Contenedor stack

- push()
- pop()
- top()

Contenedor stack

- push()
- pop()
- top()

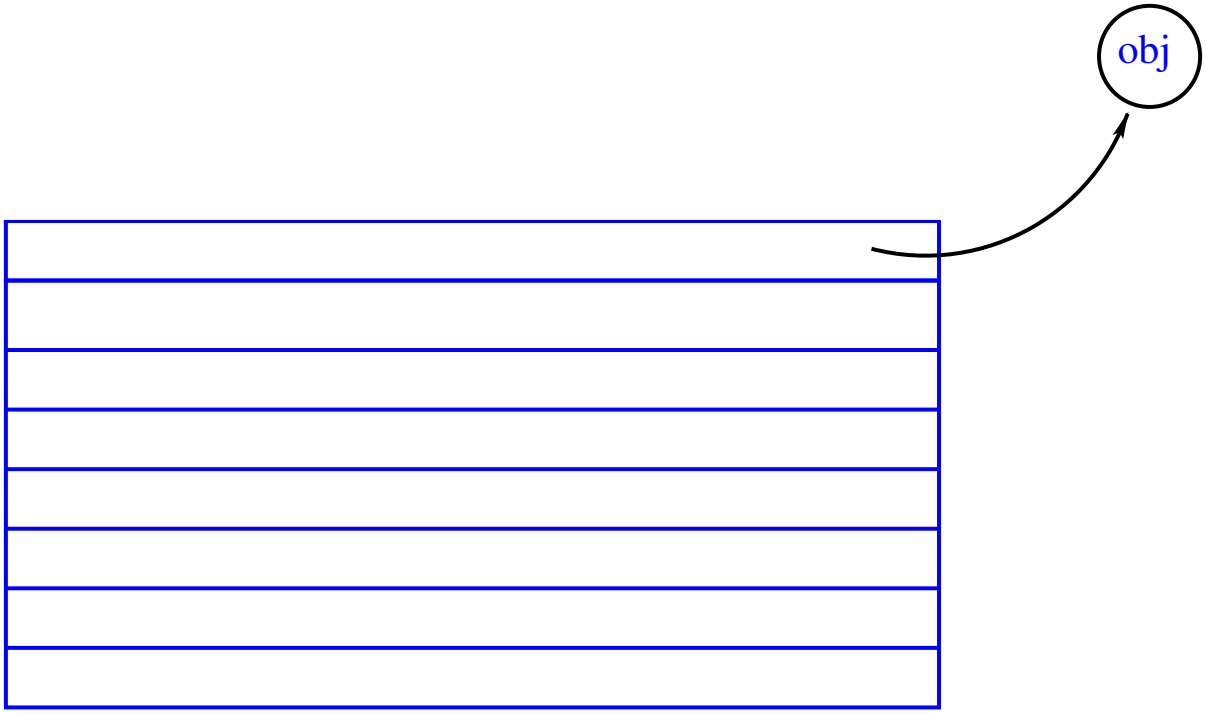
Contenedor stack

- push()
- pop()
- top()

Contenedor stack

- push()
- pop()
- top()

Contenedor stack



- push()
- pop()
- top()

Contenedor stack

- Esquema LIFO (Last In First Out).
- Métodos pop() y top() separados, ¿por qué?
- Unos métodos más: size(), empty().
- No iterador.
- Aplicaciones:
  - Secuencias de *undo* en un editor de texto.
  - Memoria para llamadas a funciones.

Contenedor stack

- Esquema LIFO (Last In First Out).
- Métodos pop() y top() separados, ¿por qué?
- Unos métodos más: size(), empty().
- No iterador.
- Aplicaciones:
  - Secuencias de *undo* en un editor de texto.
  - Memoria para llamadas a funciones.

Contenedor stack

- Esquema LIFO (Last In First Out).
- Métodos pop() y top() separados, ¿por qué?
- Unos métodos más: size(), empty().
- No iterador.
- Aplicaciones:
  - Secuencias de *undo* en un editor de texto.
  - Memoria para llamadas a funciones.

Contenedor stack

Implementación:

```
stack<int> s;
```

usa una deque por default, pero se puede usar list o vector.

- Vector

```
stack<int, vector<int> > s;
```

- Lista

```
stack<int, list<int> > s;
```

Contenedor stack

El patrón:

```
const_reference top() const {
 return c.back();
}
void push(const value_type& __x) {
 c.push_back(__x);
}
void pop() {
 c.pop_back();
}
```

Contenedor stack

Implementaciones:

Método/Secuencia	Vector	List	Deque
top()	$O(1)$	$O(1)$	$O(1)$
pop()	$O(1)^+$	$O(1)^-$	$O(1)^+$
push()	$O(1)^+$	$O(1)^-$	$O(1)^+$
size()	$O(1)$	$O(n)^*$	$O(1)$
Redim.	O(1) (am.)	-	-

* depende del compilador

Contenedor stack

Debate para saber si *list* :: *size*() debe de ser $O(1)$ (usando un contador) o $O(n)$ (recorrer la lista cada vez):

+ Optimizaciones obvias:
  - Operador ==.
  - Determinar un sentido de recorrido eficiente con resize().
- Se necesita hacer update del contador, en particular con splice.

Contenedor queue

- push()
- pop()
- back()
- front()

Contenedor queue

- push()
- pop()
- back()
- front()

Contenedor queue

- push()
- pop()
- back()
- front()

Contenedor queue

- push()
- pop()
- back()
- front()

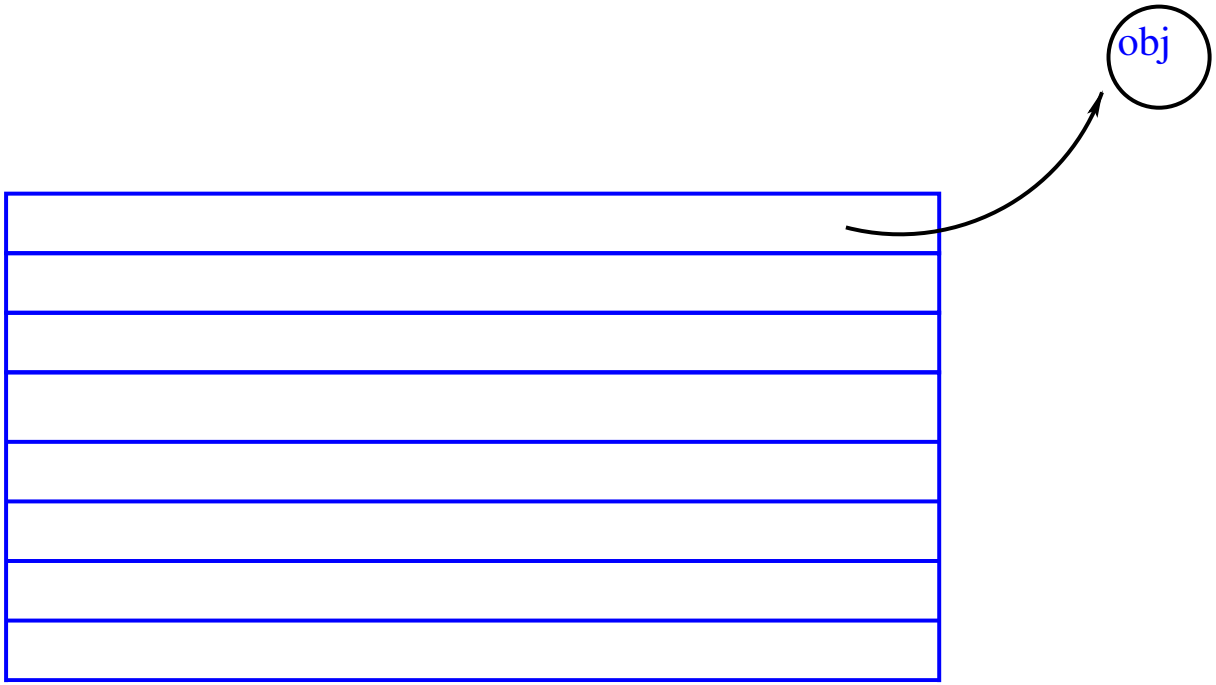
Contenedor queue

- push()
- pop()
- back()
- front()

Contenedor queue

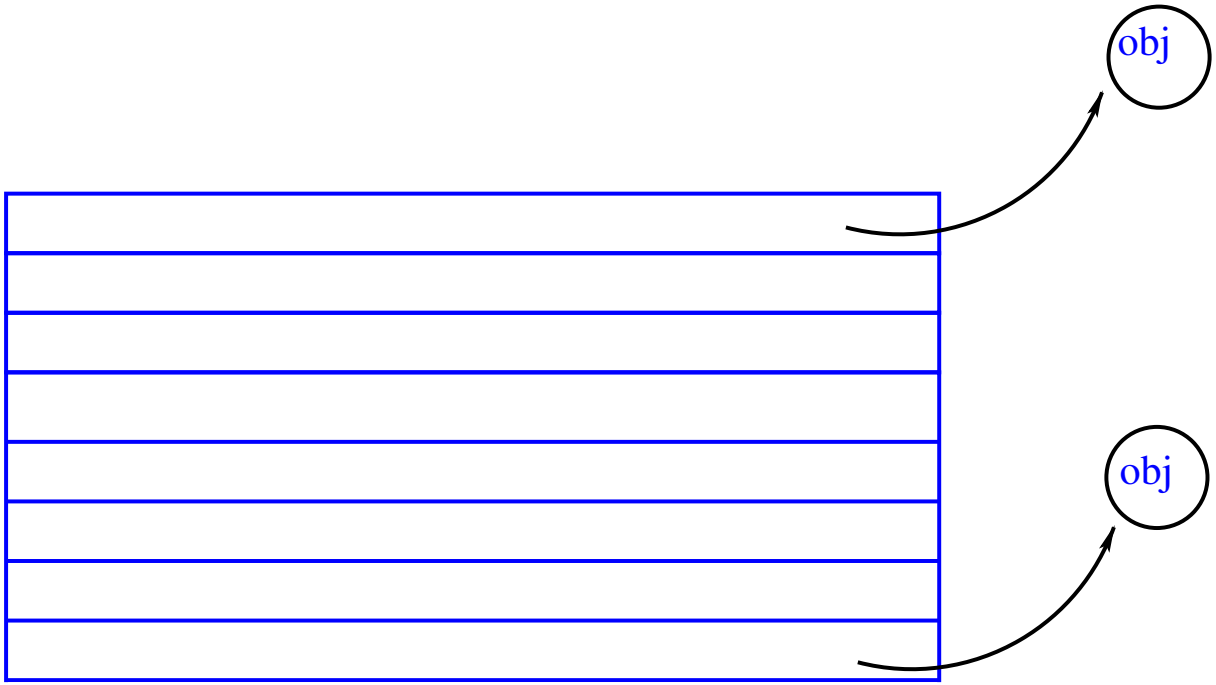
- push()
- pop()
- back()
- front()

Contenedor queue



- push()
- pop()
- back()
- front()

Contenedor queue



- push()
- pop()
- back()
- front()

Contenedor queue

- Esquema FIFO (*First In First Out*).
- Entrada por el $\mathrm{push}()$ al back, salida por el $\mathrm{pop}()$, al front.
- No iterador.
- Aplicaciones:
  - Sistemas de espera: filas de espera.
  - Sistemas de comparte de tiempo de CPU.

Contenedor queue

- Esquema FIFO (*First In First Out*).
- Entrada por el $\mathrm{push}()$ al back, salida por el $\mathrm{pop}()$, al front.
- No iterador.
- Aplicaciones:
  - Sistemas de espera: filas de espera.
  - Sistemas de comparte de tiempo de CPU.

Contenedor queue

- Esquema FIFO (*First In First Out*).
- Entrada por el $\mathrm{push}()$ al back, salida por el $\mathrm{pop}()$, al front.
- No iterador.
- Aplicaciones:
  - Sistemas de espera: filas de espera.
  - Sistemas de comparte de tiempo de CPU.

Contenedor queue

Ejemplo típico: manejar procesos!

```
class Tarea {
 string s;
public:
  Tarea(const string &title): s(title) {};
  void dolt() const {cout << "Doing " << s << endl;};
};
int main() {
  queue<Tarea> q;
  q.push(Tarea("Programacion 12"));
  q.push(Tarea("Probabilidad 8"));
  q.push(Tarea("Metodos numericos 7"));
  q.push(Tarea("Limpiar departamento"));
  while(!q.empty()) {(q.front()).dolt();q.pop(); }
}
```

Contenedor queue

Ejemplo típico: manejar procesos!

```
Doing Programacion 12
Doing Probabilidad 8
Doing Metodos numericos 7
Doing Limpiar departamento
```

Contenedor queue

Implementaciones:

Método/Secuencia	Vector	List	Deque
front()	$O(1)$	$O(1)$	$O(1)$
back()	$O(1)$	$O(1)$	$O(1)$
pop()	$O(n)$	$O(1)^-$	$O(1)^+$
push()	$O(1)^+$	$O(1)^-$	$O(1)^+$
size()	$O(1)$	$O(n)^*$	$O(1)$
Redim.	O(1) (am.)	-	-

* depende del compilador

Contenedor priority_queue

- push()
- pop()
- top()

Contenedor priority_queue

- push()
- pop()
- top()

Contenedor priority_queue

- push()
- pop()
- top()

Contenedor priority_queue

- push()
- pop()
- top()

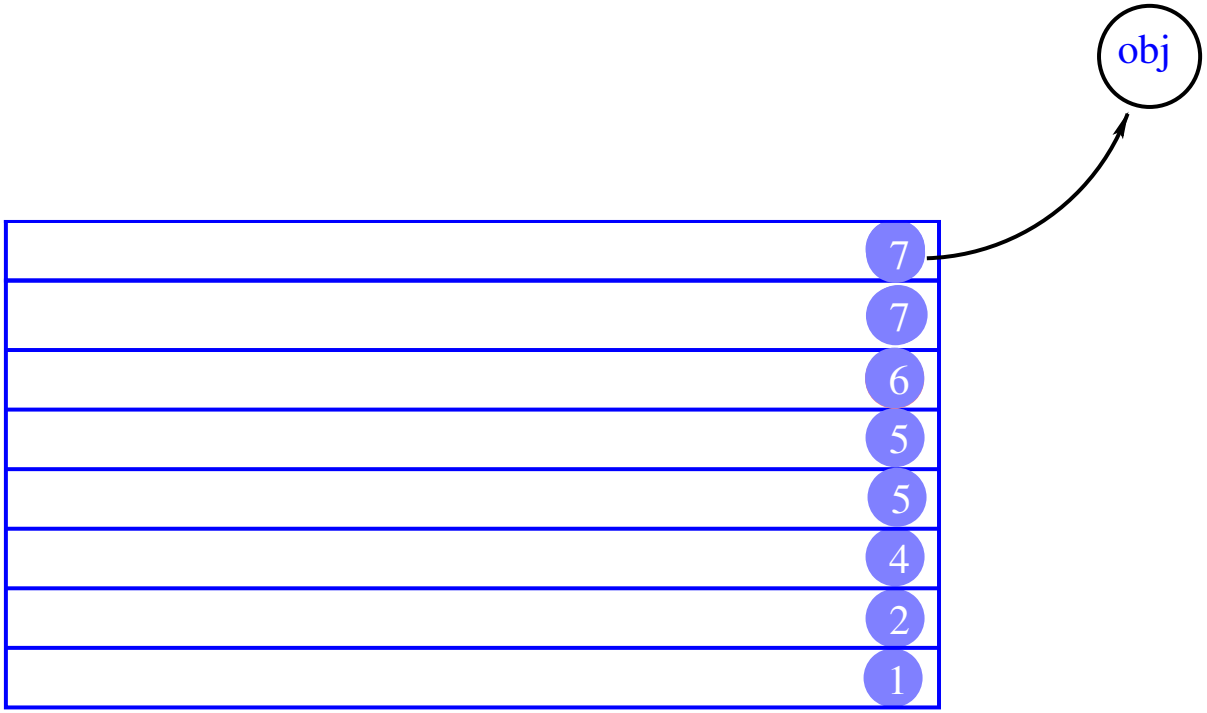
Contenedor priority_queue

- push()
- pop()
- top()

Contenedor priority_queue

- push()
- pop()
- top()

Contenedor priority_queue



- push()
- pop()
- top()

Contenedor priority_queue

- Comportamiento similar a una pila, pero con los datos ordenados por su valor.
- El $\mathrm{pop}()$ sale el objeto más prioritario de la estructura, el $\mathrm{top}()$ da una referencia hacia él.
- No iterador.
- Aplicaciones
  - Tareas de un robot ordenadas por prioridad.
  - Pacientes de un hospital.

Contenedor priority_queue

- Comportamiento similar a una pila, pero con los datos ordenados por su valor.
- El $\mathrm{pop}()$ sale el objeto más prioritario de la estructura, el $\mathrm{top}()$ da una referencia hacia él.
- No iterador.
- Aplicaciones
  - Tareas de un robot ordenadas por prioridad.
  - Pacientes de un hospital.

Contenedor priority_queue

- Comportamiento similar a una pila, pero con los datos ordenados por su valor.
- El $\mathrm{pop}()$ sale el objeto más prioritario de la estructura, el $\mathrm{top}()$ da una referencia hacia él.
- No iterador.
- Aplicaciones
  - Tareas de un robot ordenadas por prioridad.
  - Pacientes de un hospital.

Contenedor priority_queue

Ejemplo con int entre 0 y 99

```
priority_queue<int> pq;
srand(time(0));
for(int i = 0; i < 100; i++)
  pq.push(rand() % 100);
while(!pq.empty()) {
  cout << pq.top() << ' ';
  pq.pop();
}
```

Pero no es el uso mas adaptado: lo que nos gustaría es para objetos combinando un objeto (tarea) y un valor (una prioridad).

Contenedor priority_queue

En este caso, se necesita definir una clase y la métrica (operador $<$) sobre este conjunto de objetos.

```
class Tarea {
 string s;
 int priority;
public:
  Tarea(const string &title,int p=0):
        s(title),priority(p) {};
  void dolt() const {cout << "Doing " << s << endl;};
  friend bool operator<(const Tarea& a,
                        const Tarea& b) {
    return (a.priority<b.priority);
  }
};
```

Contenedor priority_queue

El uso es igual al de una pila...

```
int main() {
   priority_queue<Tarea> q;
   q.push(Tarea("Programacion 12",10));
   q.push(Tarea("Probabilidad 8",9));
   q.push(Tarea("Metodos numericos 7",8));
   q.push(Tarea("Limpiar departamento",15));
   while(!q.empty()) {(q.top()).dolt();q.pop(); }
}
```

Contenedor priority_queue

...pero el resultado muy diferente

```
Doing Limpiar departamento
Doing Programacion 12
Doing Probabilidad 8
Doing Metodos numericos 7
```

Contenedor priority_queue

Otra opción sería usar una versión completa del patrón, y redefinir less:

```
template <typename _Tp,
          typename _Sequence=vector<_Tp>,
          typename _Compare=
               less<typename _Sequence::value_type> >
    class priority_queue
```

Esta vez, se ve que la secuencia por default es un vector, usado como montículo. A priori, no se puede recorrer con iterador...

Contenedor priority_queue

...pero hay manera de acceder al contenedor secuencial subyacente (que es un vector, private, por default): el miembro c.

```
class myPq : public priority_queue<int> {
public:
  vector<int>& getVector() { return c; }
};
...
myPq pq;
for(int i = 0; i < 10; i++)
  pq.push(rand() % 100);
copy(pq.getVector().begin(), pq.getVector().end(),
     ostream_iterator<int>(cout, " "));
cout << endl;
while(!pq.empty()) {
  cout << pq.top() << " "; pq.pop();
}
```

Contenedor priority_queue

Como se ve, el orden de almacenamiento no es el orden de salida:

```
99 96 81 19 62 41 47 12 13 52
99 96 81 62 52 47 41 19 13 12
```

La estructura interna es la de un árbol (montículo) (un árbol en que todos los elementos abajo del nodo corriente son de valores inferiores al valor de este nodo. ¿Idea sobre como esta hecha la organización de los nodos en esta estructura lineal ?

Contenedor priority_queue

Las complejidades son similares a la de la pila, excepto que después de cada operación de tipo push o pop se necesita "restablecer" la estructura de montículo, y que el costo de dicha operación es $O(\log n)$.

Contenedores de bits

En varias ocasiones, se puede necesitar manipular contenedores de bits, particularmente en aplicaciones cerca del *hardware*. Existen dos contenedores adaptados a este caso:

- bitset<n>, patrón parametrizado por el numero de bits que considerar,
- vector<bool>, implementación optimizada de vector en el caso de bits,

que no tienen interfaz entre ellos...

Contenedores de bits

bitset<n>:

- Secuencia de bits, y acceso en tiempo constante a cada de los bits.
- Tamaño fijado a la implementación del patrón
- No iterador.
- Estructura atípica, con una interfaz similar a la de enteros no signados, con operaciones como & =.

Contenedores de bits

bitset<n>:

```
const bitset<12> mask(2730ul);
cout << "mask = " << mask << endl;

bitset<12> x;

cout << "Enter a 12-bit bitset in binary: ";
if (cin >> x) {
  cout << "x = " << x << endl;
  cout << "As ulong: " << x.to_ulong() << endl;
  cout << "And with mask: " << (x & mask) << endl;
  cout << "Or with mask: " << (x | mask) << endl;
}
```

Contenedores de bits

```
mask =       101010101010
Enter a 12-bit bitset in binary: 010101111001
x =          010101111001
As ulong:    1401
And with mask: 000000101000
Or with mask:  111111111011
```

Contenedores de bits

vector<bool>:

- en la mayoría de los casos, las implementaciones de bool usan un octeto entero, lo que no es optimo,
- la especialización de vector para este caso sí usa un bit por booleano,
- es extensible y tiene iteradores (pero que no pueden ser simples apuntadores a booleanos),
- mucho menos operaciones técnicas que con bitset, pero todas las de vector y un método flip() para invertir todo la secuencia de bits,
- el operador [] regresa un vector<bool>::reference, que permite operación flip().

Contenedores de bits

```
vector<bool> vb(10, true);
vector<bool>::iterator it;
for(it = vb.begin(); it != vb.end(); it++)
  cout << *it;
cout << endl;
vb.push_back(false);
ostream_iterator<bool> out(cout, "");
copy(vb.begin(), vb.end(), out);
cout << endl;
vb.flip(); // Flip all bits
copy(vb.begin(), vb.end(), out);
cout << endl;
for(size_t i = 0; i < vb.size(); i++)
  vb[i] = 0; // (Equivalent to "false")
vb[4] = true; vb[5] = 1; vb[7].flip();
copy(vb.begin(), vb.end(), out);
```

Contenedores de bits

```
// Convert to a bitset:
ostringstream os;
copy(vb.begin(), vb.end(),
   ostream_iterator<bool>(os, ""));
bitset<10> bs(os.str());
cout << "Bitset: " << endl << bs << endl;
```

Outline

Contenedor set

La filosofía de este contenedor es de almacenar datos de valores únicos, y de poder rápidamente poder determinar una relación de membrecía de un objeto con respeto a este contenedor (noción de conjunto matemático).

Para lograr este objetivo, la implementación mas común usa arboles equilibrados y por construcción ordena los datos. Operaciones de busqueda en que complejidad?

Contenedor set

Otra diferencia es que los objetos en este contenedor ya no son indexados por un index (como en los casos de vector, deque, list) sino por su valor.

Contenedor set

Ejemplo típico: crear un index para un libro

- Leer el texto del libro.
- Para cada palabra encontrada, intentar añadirla en el set.
  - Si ya esta, dejarla.
  - Si no está, añadirla al set, de tal manera que el conjunto quede ordenado y que el árbol subyacente sea equilibrado.

Contenedor set

```
char* fname = "2donq10.txt"; // Don quijote
ifstream in(fname);
set<string> wordlist;
string line;
while(getline(in, line)) {
  transform(line.begin(), line.end(), line.begin(),
            check);
  istringstream is(line);
  string word;
  while(is >> word)
    wordlist.insert(word);
}
// Output results:
copy(wordlist.begin(), wordlist.end(),
     ostream_iterator<string>(cout, "\n"));
```

Contenedor set

```
char check(char c) {
  // Only keep lower aphas
  return (isalpha(c)) ? tolower(c) : '␣';
}
```

Ahora, las palabras del Don Quijote están ordenadas y buscar una palabra dada para saber si esta tendrá complejidad logarítmica: método find(). Observar que existe también una función-algoritmo find() pero que recorre los iteradores del contenedor (complejidad lineal)

Contenedor set

Para buscar un elemento:

```
set<string>::iterator it=wordlist.find("feliz");
 if (it!=wordlist.end())
   cout << "Found before " << *(++it) << endl;
 else
   cout << "Not found" << endl;
```

```
Found before felizmente
```

Contenedores asociativos

Como su nombre lo indican, asocian llaves y valores en una estructura, con la meta de poder hacer pedidas sobre valores (objetos), en función de la llave:

- set y multiset solo contienen valores,
- map y multimap sí realizan la asociación, con el mismo tipo de estructuras.

La meta esencial de esos contenedores es de encontrar de manera eficiente la existencia de objetos: ejemplo clásico, una palabra está o no en el diccionario, y si está, cual es su definición?

Contenedores asociativos

Métodos comunes a todos:

- $\mathrm{insert}()$, para integrar nuevos objetos si su llave no está ya en el contenedor
- $\mathrm{count}()$, para contar el número de objetos que tienen una llave dada (0 o 1 en el caso de set y map, un entero positivo en el caso de $\mathrm{multiset}$ y $\mathrm{multimap}$),
- $\mathrm{find}()$ regresa un iterador sobre la posición en que se encuentra la (primera) llave dada, o $\mathrm{end}()$ sino.

Contenedores asociativos

- El contenedor $\mathrm{multimap}$ (como $\mathrm{multiset}$) tiene la propiedad de poder almacenar varios elementos de mismas llaves: un anuario telefonico por ejemplo, podría estar representado así.
- El $\mathrm{multiset}$ es útil a partir del momento que los objetos llaves sí se comparan igual pero que difieren por otros aspectos.

Contenedores asociativos

Métodos

- lower_bound(): regresa un iterador sobre el primer elemento que tiene una llave no inferior a la del parámetro.
- upper_bound(): regresa un iterador sobre el primer elemento que tiene una llave superior a la del parámetro.
- equal_range(): regresa un rango de iteradores entre los cuales la llave es igual a la del parámetro.
- En el caso de map y multimap, operador [].

Contenedores asociativos

El operador [] no se usa exactamente como lo vimos hasta ahora: toma la llave como argumento, y, si no esta esta llave en el contenedor, añade la llave (y el objeto asociado en caso de map)

```
mapped_type&
    operator[](const key_type& __k) {

      iterator __i = lower_bound(__k);
      // __i->first is greater than or equivalent to
      if (__i == end() ||
          key_comp()(__k, (*__i).first))
        __i = insert(__i, value_type(__k, mapped_type
      return (*__i).second;
    }
```

Observar que los objetos subyacentes combinan la llave (first) y el objeto (second).

Contenedores asociativos

```
template<class T1, class T2> struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair();
  pair(const T1& x, const T2& y) : first(x),
                                   second(y) {}
  // Templatized copy-constructor:
  template<class U, class V>
  pair(const pair<U, V> &p);
};
```

make_pair: para crear una nueva par (llave, valor)

Contenedores asociativos

Un ejemplo:

```
struct ltstr {
bool operator()(const char* s1,
                const char* s2) const {
    return strcmp(s1, s2) < 0;
  }
};
...
map<const char*, int, ltstr> months;
months["january"] = 31; months["february"] = 28;
months["march"] = 31;   months["april"] = 30;
months["may"] = 31;     months["june"] = 30;
months["july"] = 31;    months["august"] = 31;
months["september"] = 30; months["october"] = 31;
months["november"] = 30;  months["december"] = 31;
```

Contenedores asociativos

Un ejemplo:

```
cout << "june -> " << months["june"] << endl;
map<const char*,int,ltstr>::iterator cur
                = months.find("june");
map<const char*,int,ltstr>::iterator prev = cur;
map<const char*,int,ltstr>::iterator next = cur;
++next;
--prev;
cout << "Previous (in alphabetical order) is "
     << (*prev).first << endl;
cout << "Next (in alphabetical order) is "
     << (*next).first << endl;
```

Contenedores asociativos

Salida:

```
june -> 30
Previous (in alphabetical order) is july
Next (in alphabetical order) is march
```

Contenedores asociativos

Para usar contenedores asociativos de tipo map con algoritmos genéricos, usar por ejemplo la función inserter, y la función make_pair para inicializar nuevos objetos

```
multimap<string ,int> mm;
fill_n(inserter(mm, mm.begin()),5,
        make_pair("Test",10));
copy(mm.begin(), mm.end(),
    ostream_iterator<pair<string ,int> >(cout,"\n"));
```

que supone definido un *overload* del operador << hacia ostream, para pares.

Contenedores asociativos

También con un generador de pares aleatorias:

```
// A generator
template<typename A, typename B> class randomG {

  A miniA,maxiA;
  B miniB,maxiB;
public:
  randomG(A minA, A maxA,
          B minB, B maxB) : miniA(minA),
                            maxiA(maxA),
                            miniB(minB),
                            maxiB(maxB) {}
  std::pair<A,B> operator()() {
    return std::pair<A,B>(rand()%(maxiA-miniA)+miniA,
                          rand()%(maxiB-miniB)+min
};
```

Contenedores asociativos

```
generate_n(inserter(m, m.begin()), 10,
    randomG<int, int>(0,10,0,21));
copy(m.begin(), m.end(),
    ostream_iterator<pair<int,int> >(cout,"\n"));
```