

# Contenedores asociativos en la STL de C++

---

mat-151

# Variables y funciones estáticas de clase

```
1  class Something
2  {
3  public:
4      static int s_nValue;
5  };
6
7  int Something::s_nValue = 1;
8
9  int main()
10 {
11     Something cFirst;
12     cFirst.s_nValue = 2;
13
14     Something cSecond;
15     std::cout << cSecond.s_nValue;
16
17     return 0;
18 }
```

```
1  class Something
2  {
3  private:
4      static int s_nIDGenerator;
5      int m_nID;
6  public:
7      Something() { m_nID = s_nIDGenerator++; }
8
9      int GetID() const { return m_nID; }
10 };
11
12 int Something::s_nIDGenerator = 1;
13
14 int main()
15 {
16     Something cFirst;
17     Something cSecond;
18     Something cThird;
19
20     using namespace std;
21     cout << cFirst.GetID() << endl;
22     cout << cSecond.GetID() << endl;
23     cout << cThird.GetID() << endl;
24     return 0;
25 }
```

# Variables y funciones estáticas de clase

---

```
1  class IDGenerator
2  {
3  private:
4      static int s_nNextID;
5
6  public:
7      static int GetNextID() { return s_nNextID++; }
8  };
9
10 // We'll start generating IDs at 1
11 int IDGenerator::s_nNextID = 1;
12
13 int main()
14 {
15     for (int i=0; i < 5; i++)
16         cout << "The next ID is: " << IDGenerator::GetNextID() << endl;
17
18     return 0;
19 }
```

- Ejemplos: Pueden ser const y se inicializan afuera.
- Las funciones staticas accesan miembros estáticos

# Contenedores en la STL de C++

---

# Contenedores en la STL de C++

---

- Un **contenedor** es un **objeto que guarda** una colección de otros objetos.

# Contenedores en la STL de C++

---

- Un **contenedor** es un **objeto que guarda** una colección de otros objetos.
- Están implementados como **templates**, lo que les dá gran flexibilidad en los tipos que soportan.

# Contenedores en la STL de C++

---

- Un **contenedor** es un **objeto que guarda** una colección de otros objetos.
- Están implementados como **templates**, lo que les dá gran flexibilidad en los tipos que soportan.
- Los contenedores dan **acceso** a sus datos ya sea por **acceso directo** o a través de **iteradores**.

# Contenedores en la STL de C++

---

- Un **contenedor** es un **objeto que guarda** una colección de otros objetos.
- Están implementados como **templates**, lo que les dá gran flexibilidad en los tipos que soportan.
- Los contenedores dan **acceso** a sus datos ya sea por **acceso directo** o a través de **iteradores**.
- Los **iteradores** son objetos de referencia con **propiedades similares a los apuntadores**.



# Contenedores en la STL de C++

---

- Un **contenedor** es un **objeto que guarda** una colección de otros objetos.
- Están implementados como **templates**, lo que les dá gran flexibilidad en los tipos que soportan.
- Los contenedores dan **acceso** a sus datos ya sea por **acceso directo** o a través de **iteradores**.
- Los **iteradores** son objetos de referencia con **propiedades similares a los apuntadores**.
- Los **contenedores implementan estructuras** muy usadas en programación: arreglos dinámicos (**vector**), colas (**queue**), pilas (**stack**), montículos (**priority\_queue**), listas ligadas (**list**), árboles (**set**), arreglos asociativos (**map**)...

# Contenedores en la STL de C++

---

# Contenedores en la STL de C++

---

- Varios **contenedores** pueden **compartir funcionalidad**. La decisión de cuál usar depende de la eficiencia de estas operaciones en cada contenedor.

# Contenedores en la STL de C++

---

- Varios **contenedores** pueden **compartir funcionalidad**. La decisión de cuál usar depende de la eficiencia de estas operaciones en cada contenedor.
- Contenedores **secuenciales**:

# Contenedores en la STL de C++

---

- Varios **contenedores** pueden **compartir funcionalidad**. La decisión de cuál usar depende de la eficiencia de estas operaciones en cada contenedor.
- Contenedores **secuenciales**:
  - vector, deque, list

# Contenedores en la STL de C++

---

- Varios **contenedores** pueden **compartir funcionalidad**. La decisión de cuál usar depende de la eficiencia de estas operaciones en cada contenedor.
- Contenedores **secuenciales**:
  - vector, deque, list
- **Adaptadores** de contenedor:

# Contenedores en la STL de C++

---

- Varios **contenedores** pueden **compartir funcionalidad**. La decisión de cuál usar depende de la eficiencia de estas operaciones en cada contenedor.
- Contenedores **secuenciales**:
  - vector, deque, list
- **Adaptadores** de contenedor:
  - stack, queue, priority\_queue

# Contenedores en la STL de C++

---

- Varios **contenedores** pueden **compartir funcionalidad**. La decisión de cuál usar depende de la eficiencia de estas operaciones en cada contenedor.
- Contenedores **secuenciales**:
  - vector, deque, list
- **Adaptadores** de contenedor:
  - stack, queue, priority\_queue
- Contenedores **asociativos**:



# Contenedores en la STL de C++

---

- Varios **contenedores** pueden **compartir funcionalidad**. La decisión de cuál usar depende de la eficiencia de estas operaciones en cada contenedor.
- Contenedores **secuenciales**:
  - vector, deque, list
- **Adaptadores** de contenedor:
  - stack, queue, priority\_queue
- Contenedores **asociativos**:
  - set, multiset, map, multimap.

# Iteradores de contenedores

---

# Iteradores de contenedores

---

- Existen para la gran mayoría de los contenedores estándar, y son definidos:

# Iteradores de contenedores

---

- Existen para la gran mayoría de los contenedores estándar, y son definidos:

- **Contenedor::iterador**

# Iteradores de contenedores

---

- Existen para la gran mayoría de los contenedores estándar, y son definidos:
- **Contenedor::iterador**
- permitiendo recorrer como un apuntador los datos con los operadores **++**, **==**, **!=**, y los dos iteradores predefinidos **begin( )** y **end( )**.

# Iteradores de contenedores

---

- Existen para la gran mayoría de los contenedores estándar, y son definidos:
- **Contenedor::iterador**
- permitiendo recorrer como un apuntador los datos con los operadores **++**, **==**, **!=**, y los dos iteradores predefinidos **begin( )** y **end( )**.

```
for( Contenedor::iterador it=c.begin(); it!=c.end(); it++ )  
{  
    (*it).doSomething();  
    it->doSomething();  
}
```

# Contenedores secuenciales

---

# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.



# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.
- Usan **templates** que dependen del elemento que almacenan.

# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.
- Usan **templates** que dependen del elemento que almacenan.
- Inserción y eliminación.

# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.
- Usan **templates** que dependen del elemento que almacenan.
- Inserción y eliminación.
- Acceso aleatorio ( no todos los contenedores )

# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.
- Usan **templates** que dependen del elemento que almacenan.
- Inserción y eliminación.
- Acceso aleatorio ( no todos los contenedores )
- Pushing y popping en ambos extremos.

# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.
- Usan **templates** que dependen del elemento que almacenan.
- Inserción y eliminación.
- Acceso aleatorio ( no todos los contenedores )
- Pushing y popping en ambos extremos.
- **class vector** ( clase secuencial de uso general )

# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.
- Usan **templates** que dependen del elemento que almacenan.
- Inserción y eliminación.
- Acceso aleatorio ( no todos los contenedores )
- Pushing y popping en ambos extremos.
- **class vector** ( clase secuencial de uso general )
- **class deque** ( optimizado para insertar y eliminar elementos en los extremos, tiene acceso aleatorio )

# Contenedores secuenciales

---

- Variaciones en una **secuencia lineal de valores**.
- Usan **templates** que dependen del elemento que almacenan.
- Inserción y eliminación.
- Acceso aleatorio ( no todos los contenedores )
- Pushing y popping en ambos extremos.
- **class vector** ( clase secuencial de uso general )
- **class deque** ( optimizado para insertar y eliminar elementos en los extremos, tiene acceso aleatorio )
- **class list** ( optimizado para acceso secuencial e inserción en cualquier posición, no para acceso aleatorio )

# Contenedores secuenciales: **vector**

---



# Contenedores secuenciales: **vector**

---

- Están implementados como **arreglos** (como arreglos en C) **dinámicos**:

# Contenedores secuenciales: **vector**

---

- Están implementados como **arreglos** (como arreglos en C) **dinámicos**:
  - como los arreglos regulares sus **elementos** están **almacenados en posiciones adyacentes en la memoria**: permite usar iteradores y apuntadores.

# Contenedores secuenciales: **vector**

---

- Están implementados como **arreglos** (como arreglos en C) **dinámicos**:
  - como los arreglos regulares sus **elementos** están **almacenados en posiciones adyacentes en la memoria**: permite usar iteradores y apuntadores.
  - el aumento o disminución del tamaño del vector (contrariamente a los arreglos regulares) es manejado **automáticamente**.

# Contenedores secuenciales: **vector**

---

- Están implementados como **arreglos** (como arreglos en C) **dinámicos**:
  - como los arreglos regulares sus **elementos** están **almacenados en posiciones adyacentes en la memoria**: permite usar iteradores y apuntadores.
  - el aumento o disminución del tamaño del vector (contrariamente a los arreglos regulares) es manejado **automáticamente**.
- **Acceder** elementos individuales por su posición (índice) -  **$O(1)$**

# Contenedores secuenciales: vector

---

- Están implementados como **arreglos** (como arreglos en C) **dinámicos**:
  - como los arreglos regulares sus **elementos** están **almacenados en posiciones adyacentes en la memoria**: permite usar iteradores y apuntadores.
  - el aumento o disminución del tamaño del vector (contrariamente a los arreglos regulares) es manejado **automáticamente**.
- **Acceder** elementos individuales por su posición (índice) -  **$O(1)$**
- **Iterar** los elementos en cualquier orden -  **$O(n)$**

# Contenedores secuenciales: vector

---

- Están implementados como **arreglos** (como arreglos en C) **dinámicos**:
  - como los arreglos regulares sus **elementos** están **almacenados en posiciones adyacentes en la memoria**: permite usar iteradores y apuntadores.
  - el aumento o disminución del tamaño del vector (contrariamente a los arreglos regulares) es manejado **automáticamente**.
- **Acceder** elementos individuales por su posición (índice) -  **$O(1)$**
- **Iterar** los elementos en cualquier orden -  **$O(n)$**
- **Agregar o eliminar** elementos al final -  **$O(1)$**

# Contenedores secuenciales: vector

---

- Están implementados como **arreglos** (como arreglos en C) **dinámicos**:
  - como los arreglos regulares sus **elementos** están **almacenados en posiciones adyacentes en la memoria**: permite usar iteradores y apuntadores.
  - el aumento o disminución del tamaño del vector (contrariamente a los arreglos regulares) es manejado **automáticamente**.
- **Acceder** elementos individuales por su posición (índice) -  **$O(1)$**
- **Iterar** los elementos en cualquier orden -  **$O(n)$**
- **Agregar o eliminar** elementos al final -  **$O(1)$**
- **Internamente los vectores** - como todos los contenedores - **tienen un tamaño**, que representa el número de elementos en el vector.

# Contenedores secuenciales: **vector**

---



# Contenedores secuenciales: **vector**

---

- Los vectores también tienen **capacidad**, que determina el **espacio adicional que se puede utilizar** (para no re-dimensionar el tamaño de memoria cada vez).

# Contenedores secuenciales: **vector**

---

- Los vectores también tienen **capacidad**, que determina el **espacio adicional que se puede utilizar** (para no re-dimensionar el tamaño de memoria cada vez).
- **Re-dimensionar** un vector es una **operación costosa** porque generalmente involucra re-copiar el vector.

# Contenedores secuenciales: **vector**

---

- Los vectores también tienen **capacidad**, que determina el **espacio adicional que se puede utilizar** (para no re-dimensionar el tamaño de memoria cada vez).
- **Re-dimensionar** un vector es una **operación costosa** porque generalmente involucra re-copiar el vector.
- Se recomienda indicar explícitamente la capacidad para el vector con la función miembro: **vector::reserve**.

# Contenedores secuenciales: **vector**

---

- Los vectores también tienen **capacidad**, que determina el **espacio adicional que se puede utilizar** (para no re-dimensionar el tamaño de memoria cada vez).
- **Re-dimensionar** un vector es una **operación costosa** porque generalmente involucra re-copiar el vector.
- Se recomienda indicar explícitamente la capacidad para el vector con la función miembro: **vector::reserve**.
- En su implementación en la STL de C++ los vectores toman dos parámetros:

# Contenedores secuenciales: **vector**

---

- Los vectores también tienen **capacidad**, que determina el **espacio adicional que se puede utilizar** (para no re-dimensionar el tamaño de memoria cada vez).
- **Re-dimensionar** un vector es una **operación costosa** porque generalmente involucra re-copiar el vector.
- Se recomienda indicar explícitamente la capacidad para el vector con la función miembro: **vector::reserve**.
- En su implementación en la STL de C++ los vectores toman dos parámetros:
  - **template < class T, class Allocator = allocator<T> > class vector;**

# Contenedores secuenciales: **vector**

---

- Los vectores también tienen **capacidad**, que determina el **espacio adicional que se puede utilizar** (para no re-dimensionar el tamaño de memoria cada vez).
- **Re-dimensionar** un vector es una **operación costosa** porque generalmente involucra re-copiar el vector.
- Se recomienda indicar explícitamente la capacidad para el vector con la función miembro: **vector::reserve**.
- En su implementación en la STL de C++ los vectores toman dos parámetros:
  - **template < class T, class Allocator = allocator<T> > class vector;**
- donde **T** es el **tipo de elemento**,

# Contenedores secuenciales: **vector**

---

- Los vectores también tienen **capacidad**, que determina el **espacio adicional que se puede utilizar** (para no re-dimensionar el tamaño de memoria cada vez).
- **Re-dimensionar** un vector es una **operación costosa** porque generalmente involucra re-copiar el vector.
- Se recomienda indicar explícitamente la capacidad para el vector con la función miembro: **vector::reserve**.
- En su implementación en la STL de C++ los vectores toman dos parámetros:
  - **template < class T, class Allocator = allocator<T> > class vector;**
- donde **T** es el **tipo de elemento**,
- y **Allocator**: tipo de modelo de reserva de memoria.

# Contenedores secuenciales: **vector**

---

Funciones miembro:



# Contenedores secuenciales: **vector**

---

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

# Contenedores secuenciales: **vector**

---

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

## Iteradores:

# Contenedores secuenciales: vector

---

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

## Iteradores:

<b>begin</b>	regresa el iterador al inicio
<b>end</b>	regresa el iterador al final
<b>rbegin</b>	regresa el iterador reverso al inicio reverso
<b>rend</b>	regresa el iterador reverso al final reverso

# Contenedores secuenciales: **vector**

---

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador <b>=</b>	copia el contenido del vector

## Iteradores:

<b>begin</b>	regresa el iterador al inicio
<b>end</b>	regresa el iterador al final
<b>rbegin</b>	regresa el iterador reverso al inicio reverso
<b>rend</b>	regresa el iterador reverso al final reverso

## Capacidad:

# Contenedores secuenciales: vector

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

<b>empty</b>	prueba si el vector está vacío
<b>reserve</b>	pide un cambio en la capacidad

## Iteradores:

<b>begin</b>	regresa el iterador al inicio
<b>end</b>	regresa el iterador al final
<b>rbegin</b>	regresa el iterador reverso al inicio reverso
<b>rend</b>	regresa el iterador reverso al final reverso

## Capacidad:

<b>size</b>	regresa el tamaño
<b>max_size</b>	regresa el tamaño máximo
<b>resize</b>	cambia el tamaño
<b>capacity</b>	regresa la capacidad de almacenamiento

# Contenedores secuenciales: vector

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

## Iteradores:

<b>begin</b>	regresa el iterador al inicio
<b>end</b>	regresa el iterador al final
<b>rbegin</b>	regresa el iterador reverso al inicio reverso
<b>rend</b>	regresa el iterador reverso al final reverso

## Capacidad:

<b>size</b>	regresa el tamaño
<b>max_size</b>	regresa el tamaño máximo
<b>resize</b>	cambia el tamaño
<b>capacity</b>	regresa la capacidad de almacenamiento

<b>empty</b>	prueba si el vector está vacío
<b>reserve</b>	pide un cambio en la capacidad

## Acceso a elementos:

# Contenedores secuenciales: vector

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

## Iteradores:

<b>begin</b>	regresa el iterador al inicio
<b>end</b>	regresa el iterador al final
<b>rbegin</b>	regresa el iterador reverso al inicio reverso
<b>rend</b>	regresa el iterador reverso al final reverso

## Capacidad:

<b>size</b>	regresa el tamaño
<b>max_size</b>	regresa el tamaño máximo
<b>resize</b>	cambia el tamaño
<b>capacity</b>	regresa la capacidad de almacenamiento

<b>empty</b>	prueba si el vector está vacío
<b>reserve</b>	pide un cambio en la capacidad

## Acceso a elementos:

operador []	accede un elemento
<b>at</b>	accede un elemento
<b>front</b>	accede al primer elemento
<b>back</b>	accede al último elemento

# Contenedores secuenciales: vector

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

## Iteradores:

<b>begin</b>	regresa el iterador al inicio
<b>end</b>	regresa el iterador al final
<b>rbegin</b>	regresa el iterador reverso al inicio reverso
<b>rend</b>	regresa el iterador reverso al final reverso

## Capacidad:

<b>size</b>	regresa el tamaño
<b>max_size</b>	regresa el tamaño máximo
<b>resize</b>	cambia el tamaño
<b>capacity</b>	regresa la capacidad de almacenamiento

<b>empty</b>	prueba si el vector está vacío
<b>reserve</b>	pide un cambio en la capacidad

## Acceso a elementos:

operador []	accede un elemento
<b>at</b>	accede un elemento
<b>front</b>	accede al primer elemento
<b>back</b>	accede al último elemento

## Modificadores:



# Contenedores secuenciales: vector

## Funciones miembro:

(constructor)	constructor del vector
(destructor)	destructor del vector
operador =	copia el contenido del vector

## Iteradores:

<b>begin</b>	regresa el iterador al inicio
<b>end</b>	regresa el iterador al final
<b>rbegin</b>	regresa el iterador reverso al inicio reverso
<b>rend</b>	regresa el iterador reverso al final reverso

## Capacidad:

<b>size</b>	regresa el tamaño
<b>max_size</b>	regresa el tamaño máximo
<b>resize</b>	cambia el tamaño
<b>capacity</b>	regresa la capacidad de almacenamiento

<b>empty</b>	prueba si el vector está vacío
<b>reserve</b>	pide un cambio en la capacidad

## Acceso a elementos:

operador []	accede un elemento
<b>at</b>	accede un elemento
<b>front</b>	accede al primer elemento
<b>back</b>	accede al último elemento

## Modificadores:

<b>assign</b>	asigna un contenido al vector
<b>push_back</b>	agrega un elemento al final
<b>pop_back</b>	elimina el último elemento
<b>insert</b>	inserta elementos
<b>erase</b>	borra elementos
<b>swap</b>	intercambia el contenido de vectores
<b>clear</b>	limpia el contenido

# Contenedores secuenciales: vector

---

```
// Copy an entire file into a vector of string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

int main() {
    vector<string> v;
    ifstream in("../../main.cpp");
    string line;
    while(getline(in, line))
        v.push_back(line); // Add the line to the end
    // Add line numbers:
    for(int i = 0; i < v.size(); i++)
        cout << i << ": " << v[i] << endl;
}
```

# Contenedores secuenciales: vector

```
// Copy an entire file into a vector of string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

int main() {
    vector<string> v;
    ifstream in("../../main.cpp");
    string line;
    while(getline(in, line))
        v.push_back(line); // Add the line to the end
    // Add line numbers:
    for(int i = 0; i < v.size(); i++)
        cout << i << ": " << v[i] << endl;
}
```

```
[Session started at 2008-05-13 10:16:53 -0500.]
0: // Copy an entire file into a vector of string
1: #include <string>
2: #include <iostream>
3: #include <fstream>
4: #include <vector>
5:
6: using namespace std;
7:
8: int main() {
9:     vector<string> v;
10:     ifstream in("../../main.cpp");
11:     string line;
12:     while(getline(in, line))
13:         v.push_back(line); // Add the line to the end
14:     // Add line numbers:
15:     for(int i = 0; i < v.size(); i++)
16:         cout << i << ": " << v[i] << endl;
17: } ///:~

containers has exited with status 0.
```

# Contenedores

## secuenciales: vector

---

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

int main() {
    vector<string> words;
    ifstream in("../../main.cpp");
    string word;
    while(in >> word)
        words.push_back(word);
    for(int i = 0; i < words.size(); i++)
        cout << words[i] << endl;
}
```

# Contenedores secuenciales: vector

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

int main() {
    vector<string> words;
    ifstream in("../../main.cpp");
    string word;
    while(in >> word)
        words.push_back(word);
    for(int i = 0; i < words.size(); i++)
        cout << words[i] << endl;
}
```

```
[Session started at 2008-05-13 10:30:34 -0500.]
//
Copy
an
entire
file
into
a
vector
of
string
#include
<string>
#include
<iostream>
#include
<fstream>
#include
<vector>
using
namespace
std;
int
main()
{
    vector<string>
    words;
    ifstream
    in("../../main.cpp");
    string
    word;
    while(in
    >>
    word)
    words.push_back(word);
    for(int
    i
    =
    0;
    i
    <
    words.size();
    i++)
    cout
    <<
    words[i]
    <<
    endl;
}
containers has exited with status 0.
```

# Contenedores secuenciales: vector

---

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v;
    for(int i = 0; i < 10; i++)
        v.push_back(i);
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << ", ";
    cout << endl;
    for(int i = 0; i < v.size(); i++)
        v[i] = v[i] * 10; // Assignment
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << ", ";
    cout << endl;
}
```



# Contenedores secuenciales: vector

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v;
    for(int i = 0; i < 10; i++)
        v.push_back(i);
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << ", ";
    cout << endl;
    for(int i = 0; i < v.size(); i++)
        v[i] = v[i] * 10; // Assignment
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << ", ";
    cout << endl;
}
```

```
[Session started at 2008-05-13 10:56:34 -0500.]
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 10, 20, 30, 40, 50, 60, 70, 80, 90,

containers has exited with status 0.
```

# Contenedores secuenciales: deque

---



# Contenedores secuenciales: deque

---

- double-ended-queue.

# Contenedores secuenciales: deque

---

- **d**ouble-**e**nded-**q**ueue.
- **Elementos** individuales pueden ser **accesados por su posición** (índice).

# Contenedores secuenciales: deque

---

- **d**ouble-**e**nded-**q**ueue.
- **Elementos** individuales pueden ser **accesados por su posición** (índice).
- Se puede iterar sobre todos los elementos en cualquier orden.

# Contenedores secuenciales: deque

---

- **d**ouble-**e**nded-**q**ueue.
- **E**lementos individuales pueden ser **accesados por su posición** (índice).
- Se puede iterar sobre todos los elementos en cualquier orden.
- Los elementos pueden ser añadidos o eliminados eficientemente de los extremos -  **$O(1)$**

# Contenedores secuenciales: deque

---

- **d**ouble-**e**nded-**q**ueue.
- **Elementos** individuales pueden ser **accesados por su posición** (índice).
- Se puede iterar sobre todos los elementos en cualquier orden.
- Los elementos pueden ser añadidos o eliminados eficientemente de los extremos -  **$O(1)$**
- Los elementos **no se almacenan en espacios contiguos** de la memoria (usa muchos bloques)

# Contenedores secuenciales: deque

---

- **d**ouble-**e**nded-**q**ueue.
- **Elementos** individuales pueden ser **accesados por su posición** (índice).
- Se puede iterar sobre todos los elementos en cualquier orden.
- Los elementos pueden ser añadidos o eliminados eficientemente de los extremos -  **$O(1)$**
- Los elementos **no se almacenan en espacios contiguos** de la memoria (usa muchos bloques)
- La implementación en C++ toma dos parámetros:

# Contenedores secuenciales: deque

---

- **d**ouble-**e**nded-**q**ueue.
- **Elementos** individuales pueden ser **accesados por su posición** (índice).
- Se puede iterar sobre todos los elementos en cualquier orden.
- Los elementos pueden ser añadidos o eliminados eficientemente de los extremos -  **$O(1)$**
- Los elementos **no se almacenan en espacios contiguos** de la memoria (usa muchos bloques)
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Allocator = allocator<T> > class deque;**

# Contenedores secuenciales: deque

```
#include <iostream>
#include <deque>

using namespace std;

int main() {

    deque<string> animales;


    animales.push_back("Perro");
    animales.push_back("Gato");
    animales.push_back("Cerdo");
    animales.push_back("Mapache");

    animales[2] = "Gusano";

    for( int i=0; i<animales.size(); i++ )
    {
        cout << animales.at(i) << endl;
    }

    return 0;
}
```

Podríamos haber usado corchetes, pero esta llamada genera una *exception* cuando se está fuera de los límites.





# Contenedores secuenciales: deque

```
#include <iostream>
#include <deque>

using namespace std;

int main() {

    deque<string> animales;

    animales.push_back("Perro");
    animales.push_back("Gato");
    animales.push_back("Cerdo");
    animales.push_back("Mapache");

    animales[2] = "Gusano";


    for( int i=0; i<animales.size(); i++ )
    {
        cout << animales.at(i) << endl;
    }

    return 0;
}
```

```
[Session started at 2008-05-13 12:21:30 -0500.]
Perro
Gato
Gusano
Mapache

containers has exited with status 0.
```

Podríamos haber usado corchetes, pero esta llamada genera una *exception* cuando se está fuera de los límites.



# Contenedores secuenciales: **list**

---

# Contenedores secuenciales: list

---

- Implementadas como **listas doblemente ligadas**.

# Contenedores secuenciales: list

---

- Implementadas como **listas doblemente ligadas**.
- **Inserción y eliminación** eficiente (a diferencia de *deque*) de elementos en cualquier lugar de la secuencia (una vez encontrado, se usa insert, ver ejemplo en cplusplus)  **$O(1)$** .

# Contenedores secuenciales: list

---

- Implementadas como **listas doblemente ligadas**.
- **Inserción y eliminación** eficiente (a diferencia de *deque*) de elementos en cualquier lugar de la secuencia (una vez encontrado, se usa insert, ver ejemplo en cplusplus)  **$O(1)$** .
- **Iteración** de los elementos en orden directo o reverso  **$O(n)$** .

# Contenedores secuenciales: list

---

- Implementadas como **listas doblemente ligadas**.
- **Inserción y eliminación** eficiente (a diferencia de *deque*) de elementos en cualquier lugar de la secuencia (una vez encontrado, se usa insert, ver ejemplo en cplusplus)  **$O(1)$** .
- **Iteración** de los elementos en orden directo o reverso  **$O(n)$** .
- Son mejores para insertar, extraer y mover elementos de cualquier posición del contenedor (y en algoritmos que hacen uso intensivo de estas operaciones)

# Contenedores secuenciales: list

---

- Implementadas como **listas doblemente ligadas**.
- **Inserción y eliminación** eficiente (a diferencia de *deque*) de elementos en cualquier lugar de la secuencia (una vez encontrado, se usa insert, ver ejemplo en cplusplus)  **$O(1)$** .
- **Iteración** de los elementos en orden directo o reverso  **$O(n)$** .
- Son mejores para insertar, extraer y mover elementos de cualquier posición del contenedor (y en algoritmos que hacen uso intensivo de estas operaciones)
- No tienen acceso directo a elementos en forma aleatoria y toman memoria adicional para almacenar información sobre los elementos (links).

# Contenedores secuenciales: list

---

- Implementadas como **listas doblemente ligadas**.
- **Inserción y eliminación** eficiente (a diferencia de *deque*) de elementos en cualquier lugar de la secuencia (una vez encontrado, se usa insert, ver ejemplo en cplusplus)  **$O(1)$** .
- **Iteración** de los elementos en orden directo o reverso  **$O(n)$** .
- Son mejores para insertar, extraer y mover elementos de cualquier posición del contenedor (y en algoritmos que hacen uso intensivo de estas operaciones)
- No tienen acceso directo a elementos en forma aleatoria y toman memoria adicional para almacenar información sobre los elementos (links).
- **`template < class T, class Allocator=allocator<T> > class list;`**



# Contenedores secuenciales: list

---

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<char> charList;

    for( int i=0; i<10; i++ ){
        charList.push_front( i+65 );
    }

    list<char>::iterator current = charList.begin();

    for( current = charList.begin(); current != charList.end(); current++ ){
        cout << *current << endl;
    }
    return 0;
}
```

# Contenedores secuenciales: list

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<char> charList;

    for( int i=0; i<10; i++ ){
        charList.push_front( i+65 );
    }

    list<char>::iterator current = charList.begin();

    for( current = charList.begin(); current != charList.end(); current++ ){
        cout << *current << endl;
    }
    return 0;
}
```

[Session started at 2008-05-13 13:01:56 -0500.]

J  
I  
H  
G  
F  
E  
D  
C  
B  
A

containers has exited with status 0.

# Adaptadores de contenedores: **stack**

---

# Adaptadores de contenedores: **stack**

---

- **LIFO**

# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.

# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:

# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
- **template < class T, class Container = deque<T> > class stack;**

# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Container = deque<T> > class stack;**
- donde **T** es el tipo de elementos



# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Container = deque<T> > class stack;**
- donde **T** es el tipo de elementos
- y **Container** es el tipo de contenedor utilizado para acceder y almacenar elementos.

# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Container = deque<T> > class stack;**
- donde **T** es el tipo de elementos
- y **Container** es el tipo de contenedor utilizado para acceder y almacenar elementos.
- Aplicaciones:

# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Container = deque<T> > class stack;**
- donde **T** es el tipo de elementos
- y **Container** es el tipo de contenedor utilizado para acceder y almacenar elementos.
- Aplicaciones:
  - secuencias de undo en un editor de texto,

# Adaptadores de contenedores: **stack**

---

- **LIFO**
- Soporta las funciones **empty**, **size**, **top**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Container = deque<T> > class stack;**
- donde **T** es el tipo de elementos
- y **Container** es el tipo de contenedor utilizado para acceder y almacenar elementos.
- Aplicaciones:
  - secuencias de undo en un editor de texto,
  - memoria para llamadas a funciones.

# Adaptadores de contenedores: stack

```
#include <stack>
#include <iostream>

using namespace std;

int main()
{
    stack <int> st1, st2;
    // push data element on the stack
    st1.push(21);
    int j = st1.top();
    cout<<j<<' ';
    st1.push(9);
    j=st1.top();
    cout<<j<<' ';
    st1.push(12);
    j=st1.top();
    cout<<j<<' ';
    st1.push(31);
    j=st1.top();
    cout<<j<<' '<<endl;
    stack <int>::size_type i;
    i = st1.size();
    cout<<"El largo de la pila es: "<<i<<endl;
    i = st1.top();
    cout<<"El elemento arriba de la pila es "<<i<<endl;
    st1.pop();
    i = st1.size();
    cout<<"Despues de un pop, el tamano de la pila es "<<i<<endl;
    i = st1.top();
    cout<<"Despues de un pop, el tamano de la pila es "<<i<<endl;
    return 0;
}
```

# Adaptadores de contenedores: stack

```
#include <stack>
#include <iostream>

using namespace std;

int main()
{
    stack<int> st1, st2;
    // push data element on the stack
    st1.push(21);
    int j = st1.top();
    cout<<j<<' ';
    st1.push(9);
    j=st1.top();
    cout<<j<<' ';
    st1.push(12);
    j=st1.top();
    cout<<j<<' ';
    st1.push(31);
    j=st1.top();
    cout<<j<<' '<<endl;
    stack<int>::size_type i;
    i = st1.size();
    cout<<"El largo de la pila es: "<<i<<endl;
    i = st1.top();
    cout<<"El elemento arriba de la pila es "<<i<<endl;
    st1.pop();
    i = st1.size();
    cout<<"Despues de un pop, el tamano de la pila es "<<i<<endl;
    i = st1.top();
    cout<<"Despues de un pop, el tamano de la pila es "<<i<<endl;
    return 0;
}
```

```
[Session started at 2008-05-13 13:34:45 -0500.]
21 9 12 31
El largo de la pila es: 4
El elemento arriba de la pila es 31
Despues de un pop, el tamano de la pila es 3
Despues de un pop, el tamano de la pila es 12

containers has exited with status 0.
```

# Adaptadores de contenedores: **queue**

---

# Adaptadores de contenedores: **queue**

---

- cola **FIFO**



# Adaptadores de contenedores: `queue`

---

- cola **FIFO**
- Soporta las operaciones `empty`, `size`, `front`, `back`, `push`, `pop`.

# Adaptadores de contenedores: `queue`

---

- cola **FIFO**
- Soporta las operaciones `empty`, `size`, `front`, `back`, `push`, `pop`.
- La implementación en C++ toma dos parámetros:

# Adaptadores de contenedores: **queue**

---

- cola **FIFO**
- Soporta las operaciones **empty**, **size**, **front**, **back**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
- **template < class T, class Container = deque <T> > class queue;**

# Adaptadores de contenedores: **queue**

---

- cola **FIFO**
- Soporta las operaciones **empty**, **size**, **front**, **back**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Container = deque <T> > class queue;**
- Aplicaciones:

# Adaptadores de contenedores: **queue**

---

- cola **FIFO**
- Soporta las operaciones **empty**, **size**, **front**, **back**, **push**, **pop**.
- La implementación en C++ toma dos parámetros:
  - **template < class T, class Container = deque <T> > class queue;**
- Aplicaciones:
  - Sistema de espera: filas de espera

# Adaptadores de contenedores: `priority_queue`

---

# Adaptadores de contenedores: `priority_queue`

---

- Diseñado para que el **primer elemento** sea siempre el **más grande**.

# Adaptadores de contenedores: `priority_queue`

---

- Diseñado para que el **primer elemento** sea siempre el **más grande**.
- Solo se puede recuperar el elemento con la **prioridad mayor**.



# Adaptadores de contenedores: `priority_queue`

---

- Diseñado para que el **primer elemento** sea siempre el **más grande**.
- Solo se puede recuperar el elemento con la **prioridad mayor**.
- Para tener la estructura de orden interno de montículo se necesita tener **acceso aleatorio** a los elementos.

# Adaptadores de contenedores: `priority_queue`

---

- Diseñado para que el **primer elemento** sea siempre el **más grande**.
- Solo se puede recuperar el elemento con la **prioridad mayor**.
- Para tener la estructura de orden interno de montículo se necesita tener **acceso aleatorio** a los elementos.
- El orden se mantiene con los algoritmos **`make_heap`**, **`push_heap`** y **`pop_heap`**.

# Adaptadores de contenedores: `priority_queue`

---

- Diseñado para que el **primer elemento** sea siempre el **más grande**.
- Solo se puede recuperar el elemento con la **prioridad mayor**.
- Para tener la estructura de orden interno de montículo se necesita tener **acceso aleatorio** a los elementos.
- El orden se mantiene con los algoritmos **make\_heap**, **push\_heap** y **pop\_heap**.
- La implementación en C++ toma tres parametros:

# Adaptadores de contenedores: `priority_queue`

---

- Diseñado para que el **primer elemento** sea siempre el **más grande**.
- Solo se puede recuperar el elemento con la **prioridad mayor**.
- Para tener la estructura de orden interno de montículo se necesita tener **acceso aleatorio** a los elementos.
- El orden se mantiene con los algoritmos **make\_heap**, **push\_heap** y **pop\_heap**.
- La implementación en C++ toma tres parametros:
  - **`template < class T, class Container = vector<T>, class Compare = less<typename Container::value_type> > class priority_queue;`**

# Adaptadores de contenedores: `priority_queue`

---

- Diseñado para que el **primer elemento** sea siempre el **más grande**.
- Solo se puede recuperar el elemento con la **prioridad mayor**.
- Para tener la estructura de orden interno de montículo se necesita tener **acceso aleatorio** a los elementos.
- El orden se mantiene con los algoritmos **make\_heap**, **push\_heap** y **pop\_heap**.
- La implementación en C++ toma tres parametros:
  - **`template < class T, class Container = vector<T>, class Compare = less<typename Container::value_type> > class priority_queue;`**
- donde **T** es el tipo de elemento, **Container** es el contenedor de base y **Compare** es la clase que implementa las funciones que determinan el orden.

# Adaptadores de contenedores: `priority_queue`

---

# Adaptadores de contenedores: `priority_queue`

---

- No usan iteradores

# Adaptadores de contenedores: `priority_queue`

---

- No usan iteradores
- la función miembro `pop()` regresa el objeto prioritario mientras que `top()` solo regresa un apuntador hacia él.



# Adaptadores de contenedores: `priority_queue`

---

- No usan iteradores
- la función miembro `pop()` regresa el objeto prioritario mientras que `top()` solo regresa un apuntador hacia él.
- Aplicaciones:

# Adaptadores de contenedores: `priority_queue`

---

- No usan iteradores
- la función miembro `pop()` regresa el objeto prioritario mientras que `top()` solo regresa un apuntador hacia él.
- Aplicaciones:
  - tareas de un robot ordenadas por prioridad

# Adaptadores de contenedores: `priority_queue`

---

- No usan iteradores
- la función miembro `pop()` regresa el objeto prioritario mientras que `top()` solo regresa un apuntador hacia él.
- Aplicaciones:
  - tareas de un robot ordenadas por prioridad
  - pacientes de un hospital.

```

// Using priority_queue with deque
// Use of function less sorts the items in ascending order
typedef deque<int> intdeque;
typedef priority_queue<char, intdeque, less<int> > intprque;

// Using priority_queue with vector
// Use of function greater sorts the items in descending order
typedef vector<char> chvector;
typedef priority_queue<char, chvector, greater<char> > chprque;

int main(void){
    size_t size_q;
    intprque q;
    chprque p;

    // Insert items in the priority_queue(uses deque)
    q.push(42);
    q.push(100);
    q.push(49);
    q.push(201);

    // Output the item at the top using top()
    cout << q.top() << endl;

    // Output the size of priority_queue
    size_q = q.size();
    cout << "size of q is:" << size_q << endl;

    // Output items in priority_queue using top()
    // and use pop() to get to next item until
    // priority_queue is empty
    while (!q.empty())
    {
        cout << q.top() << endl;
        q.pop();
    }

    // Insert items in the priority_queue(uses vector)
    p.push('c');
    p.push('a');

```

```

p.push('d');
p.push('m');
p.push('h');

// Output the item at the top using top()
cout << p.top() << endl;

// Output the size of priority_queue
size_q = p.size();
cout << "size of p is:" << size_q << endl;

// Output items in priority_queue using top()
// and use pop() to get to next item until
// priority_queue is empty
while (!p.empty())
{
    cout << p.top() << endl;
    p.pop();
}
}

```

```

// Using priority_queue with deque
// Use of function less sorts the items in ascending order
typedef deque<int> intdeque;
typedef priority_queue<char, intdeque, less<int> > intprque;

// Using priority_queue with vector
// Use of function greater sorts the items in descending order
typedef vector<char> chvector;
typedef priority_queue<char, chvector, greater<char> > chprque;

int main(void){
    size_t size_q;
    intprque q;
    chprque p;

    // Insert items in the priority_queue(uses deque)
    q.push(42);
    q.push(100);
    q.push(49);
    q.push(201);

    // Output the item at the top using top()
    cout << q.top() << endl;

    // Output the size of priority_queue
    size_q = q.size();
    cout << "size of q is:" << size_q << endl;

    // Output items in priority_queue using top()
    // and use pop() to get to next item until
    // priority_queue is empty
    while (!q.empty())
    {
        cout << q.top() << endl;
        q.pop();
    }

    // Insert items in the priority_queue(uses vector)
    p.push('c');
    p.push('a');

```

```

p.push('d');
p.push('m');
p.push('h');

// Output the item at the top using top()
cout << p.top() << endl;

// Output the size of priority_queue
size_q = p.size();
cout << "size of p is:" << size_q << endl;

// Output items in priority_queue using top()
// and use pop() to get to next item until
// priority_queue is empty
while (!p.empty())
{
    cout << p.top() << endl;
    p.pop();
}
}

```

**[Session started at 2008-05-13 19:01:45 -0500.]**

```

201
size of q is:4
201
100
49
42
a
size of p is:5
a
c
d
h
m

```

**containers has exited with status 0.**

# Contenedores de bits

---

# Contenedores de bits

---

- A veces se puede necesitar manipular contenedores de bits (elementos con solo dos valores posibles 0 o 1, true y false ...), particularmente en aplicaciones que tienen que ver con el hardware.

# Contenedores de bits

---

- A veces se puede necesitar manipular contenedores de bits (elementos con solo dos valores posibles 0 o 1, true y false ...), particularmente en aplicaciones que tienen que ver con el hardware.
- La clase es muy similar a un arreglo regular pero optimiza espacio para almacenamiento de estos tipos de datos.



# Contenedores de bits

---

- A veces se puede necesitar manipular contenedores de bits (elementos con solo dos valores posibles 0 o 1, true y false ...), particularmente en aplicaciones que tienen que ver con el hardware.
- La clase es muy similar a un arreglo regular pero optimiza espacio para almacenamiento de estos tipos de datos.
- Existen dos contenedores adaptados a este caso:

# Contenedores de bits

---

- A veces se puede necesitar manipular contenedores de bits (elementos con solo dos valores posibles 0 o 1, true y false ...), particularmente en aplicaciones que tienen que ver con el hardware.
- La clase es muy similar a un arreglo regular pero optimiza espacio para almacenamiento de estos tipos de datos.
- Existen dos contenedores adaptados a este caso:
  - **bitset<n>**, patrón parametrizado por el número de bits a considerar (tamaño fijo), ejemplos de funciones: **to\_ulong** , **flip** , etc.

# Contenedores de bits

---

- A veces se puede necesitar manipular contenedores de bits (elementos con solo dos valores posibles 0 o 1, true y false ...), particularmente en aplicaciones que tienen que ver con el hardware.
- La clase es muy similar a un arreglo regular pero optimiza espacio para almacenamiento de estos tipos de datos.
- Existen dos contenedores adaptados a este caso:
  - **bitset<n>**, patrón parametrizado por el número de bits a considerar (tamaño fijo), ejemplos de funciones: **to\_ulong** , **flip** , etc.
  - **vector<bool>**, implementación optimizada de un vector en el caso de bits.

# Contenedores asociativos

---

# Contenedores asociativos

---

- Como su nombre lo indica, estos contenedores asocian llaves y valores en una sola estructura.

# Contenedores asociativos

---

- Como su nombre lo indica, estos contenedores asocian llaves y valores en una sola estructura.
- La idea de poder acceder valores (objetos) a partir de la llave.

# Contenedores asociativos

---

- Como su nombre lo indica, estos contenedores asocian llaves y valores en una sola estructura.
- La idea de poder acceder valores (objetos) a partir de la llave.
- **set** y **multiset** solo contienen valores ( en este caso son iguales que la llave )

# Contenedores asociativos

---

- Como su nombre lo indica, estos contenedores asocian llaves y valores en una sola estructura.
- La idea de poder acceder valores (objetos) a partir de la llave.
- **set** y **multiset** solo contienen valores ( en este caso son iguales que la llave )
- **map** y **multimap** realizan **asociación llave,valor** usando el mismo tipo de estructuras.



# Contenedores asociativos

---

- Como su nombre lo indica, estos contenedores asocian llaves y valores en una sola estructura.
- La idea de poder acceder valores (objetos) a partir de la llave.
- **set** y **multiset** solo contienen valores ( en este caso son iguales que la llave )
- **map** y **multimap** realizan **asociación llave,valor** usando el mismo tipo de estructuras.
- La meta esencial de estos contenedores es **probar de manera eficiente la existencia de objetos**: por ejemplo si una palabra está o no en el diccionario y si está cual es su definición.

# Contenedores asociativos

---

# Contenedores asociativos

---

- Los métodos comunes entre ellos son:

# Contenedores asociativos

---

- Los métodos comunes entre ellos son:
  - **insert()** : agrega nuevos objetos si su llave no está ya en el contenedor

# Contenedores asociativos

---

- Los métodos comunes entre ellos son:
  - **insert()** : **agrega nuevos objetos** si su llave no está ya en el contenedor
  - **count()** : **cuenta el número de objetos** que tiene una llave dada ( 0 o 1 en el caso de set y map y un entero positivo en el caso de multiset y multimap)

# Contenedores asociativos

---

- Los métodos comunes entre ellos son:
  - **insert()** : **agrega nuevos objetos** si su llave no está ya en el contenedor
  - **count()** : **cuenta el número de objetos** que tiene una llave dada ( 0 o 1 en el caso de set y map y un entero positivo en el caso de multiset y multimap)
  - **find()** : **regresa un iterador sobre la posición** en que se encuentra la primera llave dada o end() si no.

# Contenedores asociativos: **set**

---

# Contenedores asociativos: **set**

---

- **Contenedor asociativo** que almacena **elementos únicos** (las llaves)



# Contenedores asociativos: **set**

---

- **Contenedor asociativo** que almacena **elementos únicos** (las llaves)
- La idea principal es determinar rápidamente una relación de membresía de un objeto con respecto al contenedor (noción de conjunto matemático) .

# Contenedores asociativos: **set**

---

- **Contenedor asociativo** que almacena **elementos únicos** (las llaves)
- La idea principal es determinar rápidamente una relación de membresía de un objeto con respecto al contenedor (noción de conjunto matemático) .
- La **implementación más común usa árboles binarios de búsqueda auto-equilibrados** y por construcción ordena los datos.

# Contenedores asociativos: **set**

---

- **Contenedor asociativo** que almacena **elementos únicos** (las llaves)
- La idea principal es determinar rápidamente una relación de membresía de un objeto con respecto al contenedor (noción de conjunto matemático) .
- La **implementación más común usa árboles binarios de búsqueda auto-equilibrados** y por construcción ordena los datos.
- Los objetos ya no son nombrados por índice sino por su valor.

# Contenedores asociativos: set

---

- **Contenedor asociativo** que almacena **elementos únicos** (las llaves)
- La idea principal es determinar rápidamente una relación de membresía de un objeto con respecto al contenedor (noción de conjunto matemático) .
- La **implementación más común usa árboles binarios de búsqueda auto-equilibrados** y por construcción ordena los datos.
- Los objetos ya no son nombrados por índice sino por su valor.
- Diseñados para **acceder** los elementos **por medio de su llave**.

# Contenedores asociativos: set

---

- **Contenedor asociativo** que almacena **elementos únicos** (las llaves)
- La idea principal es determinar rápidamente una relación de membresía de un objeto con respecto al contenedor (noción de conjunto matemático) .
- La **implementación más común usa árboles binarios de búsqueda auto-equilibrados** y por construcción ordena los datos.
- Los objetos ya no son nombrados por índice sino por su valor.
- Diseñados para **acceder** los elementos **por medio de su llave**.
- Es útil en las operaciones de unión, intersección, diferencia, y prueba de membresía.

# Contenedores asociativos: set

---

- **Contenedor asociativo** que almacena **elementos únicos** (las llaves)
- La idea principal es determinar rápidamente una relación de membresía de un objeto con respecto al contenedor (noción de conjunto matemático) .
- La **implementación más común usa árboles binarios de búsqueda auto-equilibrados** y por construcción ordena los datos.
- Los objetos ya no son nombrados por índice sino por su valor.
- Diseñados para **acceder** los elementos **por medio de su llave**.
- Es útil en las operaciones de unión, intersección, diferencia, y prueba de membresía.
- El tipo de dato debe implementar un operador de comparación.

# Contenedores asociativos: **set**

---

# Contenedores asociativos: **set**

---

- Su implementación en C++ toma tres parámetros:



# Contenedores asociativos: **set**

---

- Su implementación en C++ toma tres parámetros:
- **template < class Key, class Compare = less<Key>, class Allocator = allocator<Key> > class set.**

# Contenedores asociativos: **set**

---

- Su implementación en C++ toma tres parámetros:
- `template < class Key, class Compare = less<Key>, class Allocator = allocator<Key> > class set.`
- donde **Key** es el **tipo de elementos llave** en el contenedor. Cada elemento en un conjunto es también su llave.

# Contenedores asociativos: **set**

---

- Su implementación en C++ toma tres parámetros:
- `template < class Key, class Compare = less<Key>, class Allocator = allocator<Key> > class set.`
- donde **Key** es el **tipo de elementos llave** en el contenedor. Cada elemento en un conjunto es también su llave.
- **Compare** es la función de comparación y regresa un bool.

# Contenedores asociativos: **set**

---

- Su implementación en C++ toma tres parámetros:
- **template < class Key, class Compare = less<Key>, class Allocator = allocator<Key> > class set.**
- donde **Key** es el **tipo de elementos llave** en el contenedor. Cada elemento en un conjunto es también su llave.
- **Compare** es la función de comparación y regresa un bool.
- **Allocator** es el objeto para definir el modelo de almacenamiento.

# Contenedores asociativos: **set**

---

# Contenedores asociativos: **set**

---

- Ejemplo: índice de un libro

# Contenedores asociativos: **set**

---

- Ejemplo: índice de un libro
  - leer el texto del libro

# Contenedores asociativos: **set**

---

- Ejemplo: índice de un libro
  - leer el texto del libro
  - para cada palabra encontrada, intentar añadirla en el set



# Contenedores asociativos: **set**

---

- Ejemplo: índice de un libro
  - leer el texto del libro
  - para cada palabra encontrada, intentar añadirla en el set
    - si ya está, dejarla

# Contenedores asociativos: **set**

---

- Ejemplo: índice de un libro
  - leer el texto del libro
  - para cada palabra encontrada, intentar añadirla en el set
    - si ya está, dejarla
    - si no está, añadirla al set de tal manera que el conjunto quede ordenado y el árbol subyacente esté equilibrado.

# Contenedores asociativos: set

```
#include <iostream>
#include <set>
#include <string>

using namespace std;

int main( ) {
    set<string> setStr;
    string s="B";

    setStr.insert(s);
    s = "So";
    setStr.insert(s);
    s = "R";
    setStr.insert(s);
    s = "Toto";
    setStr.insert(s);

    for( set<string>::const_iterator p=setStr.begin(); p!= setStr.end(); ++p)
        cout << *p << endl;
}
```

# Contenedores asociativos: set

```
#include <iostream>
#include <set>
#include <string>
```

```
using namespace std;
```

```
int main( ) {
    set<string> setStr;
    string s="B";
```

```
    setStr.insert(s);
    s = "So";
    setStr.insert(s);
    s = "R";
    setStr.insert(s);
    s = "Toto";
    setStr.insert(s);
```

```
    for( set<string>::const_iterator p=setStr.begin(); p!= setStr.end(); ++p)
        cout << *p << endl;
}
```

```
[Session started at 2008-05-14 09:52:51 -0500.]
```

```
B
```

```
R
```

```
So
```

```
Toto
```

```
containers has exited with status 0.
```

# Contenedores asociativos: multiset

---

# Contenedores asociativos: multiset

---

- El contenedor multiset tiene la propiedad de poder almacenar varios elementos con la **misma llave**.

# Contenedores asociativos: multiset

---

- El contenedor multiset tiene la propiedad de poder almacenar varios elementos con la **misma llave**.
- Su implementación en C++ toma tres parámetros

# Contenedores asociativos: multiset

---

- El contenedor multiset tiene la propiedad de poder almacenar varios elementos con la **misma llave**.
- Su implementación en C++ toma tres parámetros
- ```
template < class key, class Compare = less<key>, class  
Allocator=allocator<key> > class multiset;
```



# Contenedores asociativos: multiset

```
#include <iostream>
#include <algorithm>
#include <set>
#include <iterator>

using namespace std;

int main()
{
    /* tipo de la coleccion:
     * - se permiten duplicados
     * - los elementos son enteros
     * - orden descendiente
     */
    typedef multiset<int, greater<int> > IntSet;

    IntSet coll1;

    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterar sobre los elementos e imprimirlos
    IntSet::iterator pos;
    for( pos = coll1.begin(); pos!=coll1.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

Text

# Contenedores asociativos: multiset

```
#include <iostream>
#include <algorithm>
#include <set>
#include <iterator>

using namespace std;

int main()
{
    /* tipo de la coleccion:
     * - se permiten duplicados
     * - los elementos son enteros
     * - orden descendiente
     */
    typedef multiset<int, greater<int> > IntSet;

    IntSet coll1;

    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterar sobre los elementos e imprimirlos
    IntSet::iterator pos;
    for( pos = coll1.begin(); pos!=coll1.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

Text

```
[Session started at 2008-05-14 10:27:50 -0500.]
6 5 5 4 3 2 1
```

```
containers has exited with status 0.
```

# Contenedores asociativos: map

---

# Contenedores asociativos: map

---

- Contenedores formados de la combinación llave y valor.

# Contenedores asociativos: map

---

- Contenedores formados de la **combinación llave y valor**.
- La llave se usa para identificar de manera única al elemento mientras que el valor mapeado está asociado a la llave.

# Contenedores asociativos: map

---

- Contenedores formados de la **combinación llave y valor**.
- La llave se usa para identificar de manera única al elemento mientras que el valor mapeado está asociado a la llave.
- Un ejemplo típico de un map es la guía telefónica donde el nombre es la llave y el número telefónico es el valor mapeado.

# Contenedores asociativos: map

---

- Contenedores formados de la **combinación llave y valor**.
- La llave se usa para identificar de manera única al elemento mientras que el valor mapeado está asociado a la llave.
- Un ejemplo típico de un map es la guía telefónica donde el nombre es la llave y el número telefónico es el valor mapeado.
- Internamente los elementos del mapa están ordenados de menor a mayor valor de llave.

# Contenedores asociativos: map

---

- Contenedores formados de la **combinación llave y valor**.
- La llave se usa para identificar de manera única al elemento mientras que el valor mapeado está asociado a la llave.
- Un ejemplo típico de un map es la guía telefónica donde el nombre es la llave y el número telefónico es el valor mapeado.
- Internamente los elementos del mapa están ordenados de menor a mayor valor de llave.
- Están diseñados para ser eficientes obteniendo sus elementos por una llave.



# Contenedores asociativos: map

---

- Contenedores formados de la **combinación llave y valor**.
- La llave se usa para identificar de manera única al elemento mientras que el valor mapeado está asociado a la llave.
- Un ejemplo típico de un map es la guía telefónica donde el nombre es la llave y el número telefónico es el valor mapeado.
- Internamente los elementos del mapa están ordenados de menor a mayor valor de llave.
- Están diseñados para ser eficientes obteniendo sus elementos por una llave.
- Están implementados con árboles binarios de búsqueda auto-equilibrados.

# Contenedores asociativos: map

---

# Contenedores asociativos: `map`

---

- Las características principales de `map` son:

# Contenedores asociativos: map

---

- Las características principales de map son:
- **Valores únicos de llave:** dos elementos no pueden tener la misma llave.

# Contenedores asociativos: map

---

- Las características principales de map son:
- **Valores únicos de llave**: dos elementos no pueden tener la misma llave.
- Cada elemento se compone por un **par (llave,valor)**.

# Contenedores asociativos: map

---

- Las características principales de map son:
- **Valores únicos de llave**: dos elementos no pueden tener la misma llave.
- Cada elemento se compone por un **par (llave,valor)**.
- Los elementos **siguen una relación de orden** en todo momento.

# Contenedores asociativos: map

---

- Las características principales de map son:
- **Valores únicos de llave**: dos elementos no pueden tener la misma llave.
- Cada elemento se compone por un **par (llave,valor)**.
- Los elementos **siguen una relación de orden** en todo momento.
- En su implementación en C++ toman 4 parámetros:

# Contenedores asociativos: map

---

- Las características principales de map son:
- **Valores únicos de llave**: dos elementos no pueden tener la misma llave.
- Cada elemento se compone por un **par (llave,valor)**.
- Los elementos **siguen una relación de orden** en todo momento.
- En su implementación en C++ toman 4 parámetros:
- ```
template < class Key, class T, class Compare=less<Key>, class  
Allocator=allocator<pair<const key, T>> class map;
```



# Contenedores asociativos: map

---

- Las características principales de map son:
- **Valores únicos de llave**: dos elementos no pueden tener la misma llave.
- Cada elemento se compone por un **par (llave,valor)**.
- Los elementos **siguen una relación de orden** en todo momento.
- En su implementación en C++ toman 4 parámetros:
  - `template < class Key, class T, class Compare=less<Key>, class Allocator=allocator<pair<const key, T>> class map;`
- donde **Key** es el tipo de valores de llave, **T** es el tipo de valor mapeado.

# Contenedores asociativos: map

---

- Las características principales de map son:
- **Valores únicos de llave**: dos elementos no pueden tener la misma llave.
- Cada elemento se compone por un **par (llave,valor)**.
- Los elementos **siguen una relación de orden** en todo momento.
- En su implementación en C++ toman 4 parámetros:
  - `template < class Key, class T, class Compare=less<Key>, class Allocator=allocator<pair<const key, T>> class map;`
- donde **Key** es el tipo de valores de llave, **T** es el tipo de valor mapeado.
- **Compare** es la clase de comparación y **Allocator** es el modelo de almacenamiento.

# Contenedores asociativos: map

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    map<string, long> directory;
    directory["A"] = 1234567;
    directory["B"] = 9876543;
    directory["C"] = 3459876;

    string name = "A";

    if (directory.find(name) != directory.end())
        cout << "The phone number for " << name
              << " is " << directory[name] << "\n";
    else
        cout << "Sorry, no listing for " << name << "\n";
    return 0;
}
```

# Contenedores asociativos: map

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
```

```
int main()
{
    map<string, long> directory;
    directory["A"] = 1234567;
    directory["B"] = 9876543;
    directory["C"] = 3459876;
```

```
    string name = "A";
```

```
    if (directory.find(name) != directory.end())
        cout << "The phone number for " << name
              << " is " << directory[name] << "\n";
    else
        cout << "Sorry, no listing for " << name << "\n";
    return 0;
}
```

```
[Session started at 2008-05-14 10:50:25 -0500.]
The phone number for A is 1234567
```

```
containers has exited with status 0.
```

# Contenedores asociativos: **multimap**

---

- Igual a un map pero soporta llaves duplicadas.

```

#include <iostream>
#include <map>
#include <cstring>

using namespace std;

class Name {
    char str[40];
public:
    Name() {
        strcpy(str, "");
    }
    Name(char *s) {
        strcpy(str, s);
    }
    char *get() {
        return str;
    }
};

// Must define less than relative to Name objects.
bool operator<(Name a, Name b)
{
    return strcmp(a.get(), b.get()) < 0;
}

class Number {
    char str[80];
public:
    Number() {
        strcpy(str, "");
    }
    Number(char *s) {
        strcpy(str, s);
    }
    char *get() {
        return str;
    }
};

int main()
{
    map<Name, Number> directory;

    directory.insert(pair<Name, Number>(Name("T"), Number("555-4444")));
    directory.insert(pair<Name, Number>(Name("C"), Number("555-3333")));
    directory.insert(pair<Name, Number>(Name("J"), Number("555-2222")));
    directory.insert(pair<Name, Number>(Name("R"), Number("555-1111")));

    char str[80] = "T";

    map<Name, Number>::iterator p;

    p = directory.find(Name(str));
    if(p != directory.end())
        cout << "Phone number: " << p->second.get();
    else
        cout << "Name not in directory.\n";

    return 0;
}

```

**[Session started at 2008-05-14 12:38:58 -0500.]**  
**Phone number: 555-4444**  
**containers has exited with status 0.**

# Contenedores asociativos: extensiones

---

# Contenedores asociativos: extensiones

---

- De manera similar a set, multiset, map o multimap, se implementaron `hash_set`, `hash_multiset`, `hash_map` y `hash_multimap` con tablas de hash.



# Contenedores asociativos: extensiones

---

- De manera similar a set, multiset, map o multimap, se implementaron **hash\_set, hash\_multiset, hash\_map y hash\_multimap** con tablas de hash.
- En este caso las llaves no están ordenadas pero debe existir una función de hash para cada tipo de llave.

# Contenedores asociativos: extensiones

---

- De manera similar a set, multiset, map o multimap, se implementaron **hash\_set, hash\_multiset, hash\_map y hash\_multimap** con tablas de hash.
- En este caso las llaves no están ordenadas pero debe existir una función de hash para cada tipo de llave.
- Estos contenedores no son parte de la librería estándar de C++ pero están incluidos en las extensiones de STL de SGI y también en librerías muy usadas tales como la GNU C++ Library usando el `__gnu_cxx` namespace.

# Contenedores asociativos: extensiones

---

- De manera similar a set, multiset, map o multimap, se implementaron **hash\_set, hash\_multiset, hash\_map y hash\_multimap** con tablas de hash.
- En este caso las llaves no están ordenadas pero debe existir una función de hash para cada tipo de llave.
- Estos contenedores no son parte de la librería estándar de C++ pero están incluidos en las extensiones de STL de SGI y también en librerías muy usadas tales como la GNU C++ Library usando el `__gnu_cxx` namespace.
- Se espera que sean agregadas en la STL con los nombres de **unordered\_set, unordered\_multiset, unordered\_map y unordered\_multimap**.



# Referencias

---

- Thinking in C++ Vol. 1. Eckel, Bruce.
- <http://www.cplusplus.com/reference/stl>
- <http://www.java2s.com/Code/Cpp/CatalogCpp.htm>
- <http://www.mochima.com/tutorials/STL.html>
- <http://www.sgi.com/tech/stl/HashedAssociativeContainer.html>