LAS TABLAS HASH EN LA STL

Tipo `unordered_map` → key_type
→ mapped_type

Contenedor asociativo

Elementos (clave, valor) → clave : T1 (Key)
→ valor : T2 (T)

claves: no repetidas y usadas para buscar

Usa tablas hash para organizarse

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

Iteradores

```
unordered_map<Key, T>::iterator it;
```

(*it).first → clave [it → first]
(*it).second → valor [it → second]
(*it) → el par de tipo pair<const key, T>

unordered_map::at

mapped_type& at (const key_type& k);

(devuelve una referencia al valor correspondiente al elemento con clave k en el unordered_map)

```
#include <iostream>
#include <string>
#include <unordered_map>

int main()
{
  std::unordered_map<std::string, int> mymap =
    {{"Marte", 3000},
    {"saturno", 60000},
    {"Jupiter", 70000}};

  mymap.at("Marte") = 3396;
  mymap.at("saturno") += 272;
  return 0;
}
```

Funciones miembro

- constructor por defecto / destructor
- empty
- operator []
- at
- find
- count
- emplace (construye e inserta un elemento)
- insert
- erase
- clear
- swap
- bucket_count
- bucket_size
- load_factor
- rehash
- reserve
- hash_function
- begin / cbegin
- end / cend

Iteradores

iterador sobre el contenedor

```
iterator begin()
const_iterator begin() const
(analogo end())
```

Iterador sobre la cubeta

```
local_iterator begin(size_type n)
const local_iterator begin(size_type n) const
(analogo end())
```

Devuelven respectivamente un iterador al primer elemento del contenedor unordered_map o al primer elemento de la cubeta n en el unordered_map

Ejemplo de iteracion

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main ()
{
  unordered_map < string, string > mymap;

  mymap = {{"Australia", "Canberra"},
           {"USA", "Washington"},
           {"Francia", "Paris"}};

  cout << "mymap contiene: ";
  unordered_map <string, string> :: iterator it;
  for (it = mymap.begin(); it != mymap.end(); ++it)
      cout << " " << it -> first << ":" << it -> second;
  cout << endl;
  cout << "Las cubetas de mymap contienen: \n";
  for (unsigned int i = 0; i < mymap.bucket_count(); ++i){
      cout << "Cubeta #" << i << " contiene:";
      for (local_it = mymap.begin(i); local_it !=
                  mymap.end(i); ++local_it)
          cout << " " << local_it -> first << ":" <<
                  local_it -> second;
      cout << endl;
  }
}
```

size_type bucket (const Key_type k) const;

Devuelve la cubeta donde sería localizada la clave k

Si x fuera un elemento del mymap entonces

~~mymap.bucket(x.first)~~

devolvería la cubeta en que está la clave asociada a x

size_type bucket_count() const;

Devuelve el número de cubetas en el contenedor

size_type bucket_size (size_type n) const;

Devuelve el número de elementos de la cubeta n en el contenedor unordered_map

template <class... Args>

pair <iterator, bool> emplace (Args&&... args);

Inserta un nuevo elemento en el unordered_map si la clave es única. El nuevo elemento se construye usando los argumentos que se le pasan al constructor

```
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
  unordered_map <string, string> mymap;

  mymap.emplace ("NCC-1701", "Kirk");
  mymap.emplace ("NCC-1702", "Picard");
  mymap.emplace ("NCC-74656", "Janeway");

  cout << "mymap contiene:" << endl;
```

unordered_map::erase

Por posición:

iterator erase (const iterator position);

Por clave:

size_type erase (const key_type& k);

Por rango:

iterator erase (const_iterator first,
const_iterator last);

position: iterador apuntando al elemento que se quiere borrar en el unordered_map

k: clave del elemento a borrar

first, last: iteradores especifican el rango de elementos a borrar [first, second) (todos los elementos entre first y second incluyendo el elemento apuntado por first pero no el apuntado por second

iterator find (const Key_type & k);

const iterator find (const Key_type & k) const;

Busca el elemento con clave k y devuelve el iterador dónde está

float load_factor () const;

Devuelve el factor de carga del contenedor (razón entre el nº elementos y el nº cubetas)

$$load\_factor = \frac{size}{bucket\_count}$$

El contenedor aumenta de forma automática el nº cubetas para mantener el factor de carga por debajo de un umbral especifico (max_load_factor), provocando un rehashing cada vez que se necesite una expansión. Por defecto el max_load_factor es 1.0

size_type max_bucket_count() const;

Devuelve el máximo número de cubetas que el unordered_map puede tener

float max_load_factor() const;

Devuelve el factor de carga del unordered_map

void max_load_factor(float z);

Hace que z sea el nuevo máximo factor de carga del unordered_map

size_type max_size() const;

Devuelve el máximo número de elementos que el unordered_map puede albergar

size_type size() const;

Devuelve el número de elementos del unordered_map

swap / clear

Como siempre

mapped_type & operator[] (const Key_type & k);

mapped_type & operator[] (Key_type && k);

---

si k coincide con alguna clave en el unordered_map, devuelve una referencia al correspondiente valor que tiene asociado

void rehash (size_type n);

Pone el número de cubetas a n (debe ser mayor que el actual ni cubetas). Provoca un realojamiento de todas las claves

void reserve (size_type n);

Cambia el ni cubetas del contenedor (bucket_count) al que sea necesario para contener al menos n elementos

Si n > bucket_count(actual) * max_load_factor se fuerza un rehashing.

hasher hash_function() const;

Devuelve el objeto función hash que usa el unordered_map
(existe una por defecto y puede cambiarse pasando un puntero en el constructor del unordered_map)

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
typedef unordered_map<string, string> stringmap;
int main()
{
  stringmap mymap;
  stringmap::hasher fn = mymap.hash_function();
  cout << "this": " << fn("this") << endl;
  cout << "thin": " << fn("thin") << endl;
  return 0;
}
```

this: 671344778
thin: 322385291