STL:Diccionario

Generated by Doxygen 1.9.2

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 data< T, U > Struct Template Reference

Tipo elemento que define el diccionario. T es el tipo de dato asociado a una clave que no se repite (DNI p.ej.) y list es una lista de datos (string p.ej) asociados a la clave de tipo T. El diccionario está ordenado de menor a mayor clave.

`#include <diccionario.h>`

Public Attributes

- T clave
- list< U > info_asoci

3.1.1 Detailed Description

template<class T, class U>
struct data< T, U >

Tipo elemento que define el diccionario. T es el tipo de dato asociado a una clave que no se repite (DNI p.ej.) y list es una lista de datos (string p.ej) asociados a la clave de tipo T. El diccionario está ordenado de menor a mayor clave.

Definition at line 15 of file diccionario.h.

3.1.2 Member Data Documentation

3.1.2.1 clave

```
template<class T , class U >
T data< T, U >::clave
```

Definition at line 16 of file diccionario.h.

3.1.2.2 info_asoci

```
template<class T , class U >
list<U> data< T, U >::info_asoci
```

Definition at line 17 of file diccionario.h.

The documentation for this struct was generated from the following file:

- include/diccionario.h

3.2 Diccionario< T, U > Class Template Reference

Un diccionario es una lista de datos de los definidos anteriormente. Cuidado porque se manejan listas de listas. Se añaden 2 funciones privadas que hacen más facil la implementación de algunos operadores o funciones de la parte pública. Copiar copia un diccioario en otro y borrar elimina todos los elementos de un diccionario. La implementación de copiar puede hacerse usando iteradores o directamente usando la función assign.

```
#include <diccionario.h>
```

Public Member Functions

- Diccionario ()

  Constructor por defecto.
- Diccionario (const Diccionario &D)

  Constructor de copias.
- ~Diccionario ()

  Destructor.
- Diccionario< T, U > & operator= (const Diccionario< T, U > &D)

  Operador de asignación.
- bool Esta_Clave (const T &p, typename list< data< T, U > >::iterator &it_out)

  Busca la clave p en el diccionario. Si está devuelve un iterador a dónde está clave. Si no está, devuelve end() y deja el iterador de salida apuntando al sitio dónde debería estar la clave.
- void Insertar (const T &clave, const list< U > &info)

  Inserta un nuevo registro en el diccionario. Lo hace a través de la clave e inserta la lista con toda la información asociada a esa clave. Si el diccionario no estuviera ordenado habría que usar la función sort()
- void AddSignificado_Palabra (const U &s, const T &clave)

  Añade una nueva informacion asocida a una clave que está en el diccionario. la nueva información se inserta al final de la lista de información. Si no esta la clave la inserta y añade la informacion asociada.
- bool removeSignificados_Palabra (const T &clave)

  Elimina todos los significados de una palabra a partir de la clave Funcion extra "accidental" en el proceso de remove↩ Palabra.
- bool removePalabra (const T &clave)

  Elimina la palabra y sus definiciones del diccionario.
- Diccionario operator+ (const Diccionario &d)

  Une dos diccionarios devolviendo la union en un auxiliars.
- Diccionario getElementosEntreClaves (const T &clavea, const T &claveb)

  Devuelve los elementos entre las 2 claves dadas.
- list< U > getInfo_Asoc (const T &p)

Devuelve la información (una lista) asociada a una clave p. Podrían haberse definido operator[] como data<T,U> & operator[ ](int pos){ return datos.at(pos);} const data<T,U> & operator[ ](int pos)const { return datos.at(pos);}.

- int size () const

  Devuelve el tamaño del diccionario.
- list< data< T, U > >::iterator begin ()

  Funciones begin y end asociadas al diccionario.
- list< data< T, U > >::iterator end ()

  Devuelve el iterador fin del diccionario.
- list< data< T, U > >::const_iterator begin () const

  Devuelve el iterador inicio del diccionario.
- list< data< T, U > >::const_iterator end () const

  Devuelve el iterador fin del diccionario.

3.2.1 Detailed Description

template<class T, class U>
class Diccionario< T, U >

Un diccionario es una lista de datos de los definidos anteriormente. Cuidado porque se manejan listas de listas. Se añaden 2 funciones privadas que hacen más facil la implementación de algunos operadores o funciones de la parte pública. Copiar copia un diccioario en otro y borrar elimina todos los elementos de un diccionario. La implementación de copiar puede hacerse usando iteradores o directamente usando la función assign.

Definition at line 40 of file diccionario.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Diccionario() [1/2]

```
template<class T , class U >
Diccionario< T, U >::Diccionario ( )  [inline]
```

Constructor por defecto.

Definition at line 67 of file diccionario.h.

```
00067 :datos(list<data<T,U> >()){}
```

3.2.2.2 Diccionario() [2/2]

```
template<class T , class U >
Diccionario< T, U >::Diccionario (
            const Diccionario< T, U > & D )  [inline]
```

Constructor de copias.

Parameters

D	Diccionario a copiar

Definition at line 73 of file diccionario.h.

```
00073                                           {
00074                 Copiar(D);
00075         }
```

3.2.2.3 ~Diccionario()

```
template<class T , class U >
Diccionario< T, U >::~Diccionario ( )  [inline]
```

Destructor.

Definition at line 80 of file diccionario.h.

```
00080 {}
```

3.2.3 Member Function Documentation

3.2.3.1 AddSignificado_Palabra()

```
template<class T , class U >
void Diccionario< T, U >::AddSignificado_Palabra (
            const U & s,
            const T & clave )  [inline]
```

Añade una nueva informacion asocida a una clave que está en el diccionario. la nueva información se inserta al final de la lista de información. Si no esta la clave la inserta y añade la informacion asociada.

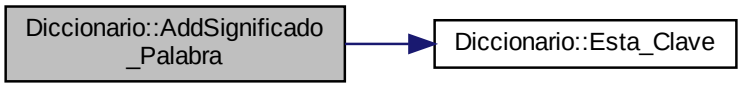
Parameters

s	La definicion a añadir
clave	La palabra a la que queremos añadir la definicion s

Definition at line 160 of file diccionario.h.

```
00160                                                                   {
00161                 typename list<data<T,U> >::iterator it;
00162
00163                 if (!Esta_Clave(clave,it)){
00164                      data<T,U> p;
00165                      p.clave = clave;
00166                      datos.insert(it,p);
00167                 }
00168
00169                  //Insertamos el siginificado al final
00170                     (*it).info_asoci.insert((*it).info_asoci.end(),s);
00171         }
```

Here is the call graph for this function:



3.2.3.2 begin() [1/2]

```
template<class T , class U >
list< data< T, U > >::iterator Diccionario< T, U >::begin ( )  [inline]
```

Funciones begin y end asociadas al diccionario.

Devuelve el iterador inicio del diccionario

Returns

un iterador que apunta al inicio del diccionario

Definition at line 310 of file diccionario.h.

```
00310                                                {
00311                 return datos.begin();
00312           }
```

3.2.3.3 begin() [2/2]

```
template<class T , class U >
list< data< T, U > >::const_iterator Diccionario< T, U >::begin ( ) const  [inline]
```

Devuelve el iterador inicio del diccionario.

Returns

un iterador no modificable que apunta al inicio del diccionario

Definition at line 325 of file diccionario.h.

```
00325                                                      {
00326                 return datos.begin();
00327           }
```

3.2.3.4 end() [1/2]

```
template<class T , class U >
list< data< T, U > >::iterator Diccionario< T, U >::end ( )  [inline]
```

Devuelve el iterador fin del diccionario.

Returns

un iterador que apunta al final del diccionario

Definition at line 317 of file diccionario.h.

```
00317                                                {
00318                 return datos.end();
00319         }
```

3.2.3.5 end() [2/2]

```
template<class T , class U >
list< data< T, U > >::const_iterator Diccionario< T, U >::end ( ) const  [inline]
```

Devuelve el iterador fin del diccionario.

Returns

un iterador no modificable que apunta al final del diccionario

Definition at line 333 of file diccionario.h.

```
00333                                                      {
00334                 return datos.end();
00335         }
```

3.2.3.6 Esta_Clave()

```
template<class T , class U >
bool Diccionario< T, U >::Esta_Clave (
            const T & p,
            typename list< data< T, U > >::iterator & it_out )  [inline]
```

Busca la clave p en el diccionario. Si está devuelve un iterador a dónde está clave. Si no está, devuelve end() y deja el iterador de salida apuntando al sitio dónde debería estar la clave.

Parameters

p	La palabra a buscar
it_out	El iterador de salida que apunta al struct 'data' donde se encuentre la palabra pasada por parametro

Returns

true si la encuentra, false si no la encuentra

Definition at line 104 of file diccionario.h.

```
00104                                                                                {
00105
00106               if (datos.size()>0){
00107
00108                   typename list<data<T,U> >::iterator it;
00109
00110                   for (it=datos.begin(); it!=datos.end() ;++it){
00111                     if ((*it).clave==p) {
00112                           it_out=it;
00113                           return true;
00114                     }
00115                     else if ((*it).clave>p){
00116                           it_out=it;
00117                           return false;
00118                     }
00119
00120                   }
00121
00122                   it_out=it;
00123                   return false;
00124               }
00125               else {
00126                     it_out=datos.end();
00127                     return false;
00128               }
00129          }
```

3.2.3.7 getElementosEntreClaves()

```
template<class T , class U >
Diccionario Diccionario< T, U >::getElementosEntreClaves (
            const T & clavea,
            const T & claveb )  [inline]
```

Devuelve los elementos entre las 2 claves dadas.

Parameters

clavea	La clave inferior
claveb	La clave superior

Returns

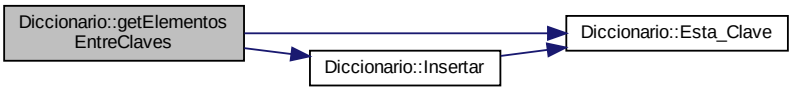
D Un nuevo diccionario con las claves intermedias con sus definiciones

Definition at line 240 of file diccionario.h.

```
00240                                                                                {
00241              Diccionario<T,U> D;
00242              typename list<data<T,U»::iterator ita, itb;
00243              bool terminar = false;
00244              int disa,disb;
00245              if (Esta_Clave(clavea, ita) && Esta_Clave(claveb, itb)){
00246                  disa = distance(datos.begin(), ita);
00247                  disb = distance(datos.begin(), itb);
00248                  if(disa <= disb){ //Si clave_a esta antes o es igual que clave_b
00249                      while (ita != itb){
00250                          D.Insertar((*ita).clave, (*ita).info_asoci);
00251                          ita++;
00252                          cout « (*ita).clave « " y " « (*itb).clave « endl;
00253                      }
00254                      D.Insertar((*ita).clave, (*ita).info_asoci); //Añadimos la clave b
```

```
00255                 }
00256                 else{ //Si clave_b esta antes que clave_a
00257                     while (itb != ita){
00258                         D.Insertar((*itb).clave, (*itb).info_asoci);
00259                         itb++;
00260                         cout « (*itb).clave « " y " « (*itb).clave « endl;
00261                     }
00262                     D.Insertar((*itb).clave, (*itb).info_asoci); //Añadimos la clave b
00263                 }
00264 
00265             }
00266             else
00267                 cerr « "Una de las claves introducidas no existe" « endl;
00268 
00269             return D;
00270         }
```

Here is the call graph for this function:



3.2.3.8 getInfo_Asoc()

```
template<class T , class U >
list< U > Diccionario< T, U >::getInfo_Asoc (
             const T & p )  [inline]
```

Devuelve la información (una lista) asociada a una clave p. Podrían haberse definido operator[] como data<T,U> & operator[ ](int pos){ return datos.at(pos);} const data<T,U> & operator[ ](int pos)const { return datos.at(pos);}.

Parameters

p	La palabra de la cual se busca su lista de definiciones

Returns

La lista de definiciones de la palabra si existe, devuelve una vacia en otro caso

Definition at line 280 of file diccionario.h.

```
00280                                           {
00281                 typename list<data<T,U> >::iterator it;
00282 
00283                 if (!Esta_Clave(p,it)){
00284                  return list<U>();
00285                 }
00286                 else{
00287                  return (*it).info_asoci;
00288                 }
00289         }
```

Here is the call graph for this function:

3.2.3.9 Insertar()

```
template<class T , class U >
void Diccionario< T, U >::Insertar (
            const T & clave,
            const list< U > & info )  [inline]
```

Inserta un nuevo registro en el diccionario. Lo hace a través de la clave e inserta la lista con toda la información asociada a esa clave. Si el diccionario no estuviera ordenado habría que usar la función sort()

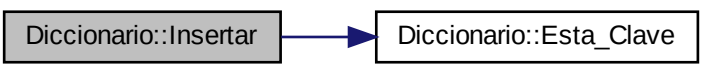
Parameters

clave	La palabra a insertar
info	La lista de definiciones a insertar

Definition at line 138 of file diccionario.h.

```
00138                                                          {
00139
00140               typename list<data<T,U> >::iterator it;
00141
00142               if (!Esta_Clave(clave,it)){
00143                     data<T,U> p;
00144                     p.clave = clave;
00145                     p.info_asoci=info;
00146
00147                     datos.insert(it,p);
00148
00149               }
00150
00151          }
```

Here is the call graph for this function:



3.2.3.10 operator+()

```
template<class T , class U >
Diccionario Diccionario< T, U >::operator+ (
            const Diccionario< T, U > & d )  [inline]
```

Une dos diccionarios devolviendo la union en un auxiliars.

Parameters

d	Diccionario a sumar, añadiendo palabras y definiciones nuevas

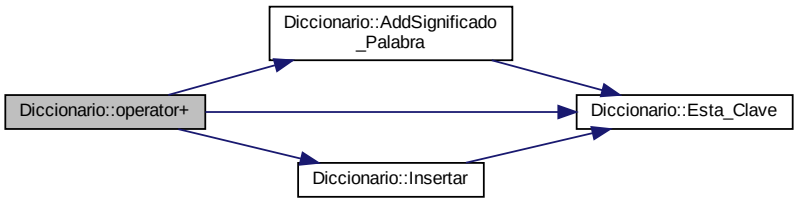
Returns

aux Un diccionario que contiene ambos diccionarios

Definition at line 211 of file diccionario.h.

```
00211                                                   {
00212             Diccionario aux(*this);
00213             typename list<data<T,U»::iterator it2;
00214             typename list<data<T,U»::const_iterator it;;
00215             for (it=d.datos.begin();it!=d.datos.end();++it){
00216                 if(aux.Esta_Clave((*it).clave,it2)){ //mete las definiciones que no existan de la
      palabra
00217                     typename list<U>::const_iterator itinfod, itinfoaux;
00218                     bool existedef = false;
00219                     for (itinfod=(*it).info_asoci.begin();itinfod !=
      (*it).info_asoci.end();++itinfod){
00220                         existedef = false;
00221                         for(itinfoaux=(*it2).info_asoci.begin();itinfoaux != (*it2).info_asoci.end()
      && !existedef;++itinfoaux)
00222                             if((*itinfoaux) == (*itinfod))
00223                                 existedef = true;
00224                         if(!existedef)
00225                             aux.AddSignificado_Palabra((*itinfod),(*it).clave);
00226                     }
00227                 }
00228                 else //Si no existe la palabra, la mete
00229                     aux.Insertar((*it).clave,(*it).info_asoci);
00230             }
00231             return aux;
00232         }
```

Here is the call graph for this function:



3.2.3.11 operator=()

```
template<class T , class U >
Diccionario< T, U > & Diccionario< T, U >::operator= (
            const Diccionario< T, U > & D )  [inline]
```

Operador de asignación.

Parameters

D	Diccionario a ser asignado

Returns

this el propio objeto

Definition at line 87 of file diccionario.h.

```
00087                                                                   {
00088                 if (this!=&D){
00089                       Borrar();
00090                       Copiar(D);
00091                 }
00092                 return *this;
00093           }
```

3.2.3.12 removePalabra()

```
template<class T , class U >
bool Diccionario< T, U >::removePalabra (
            const T & clave )  [inline]
```

Elimina la palabra y sus definiciones del diccionario.

Parameters

clave	La palabra a eliminar

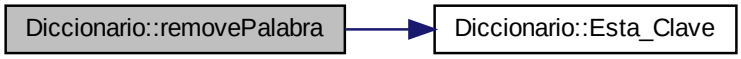
Returns

true si borra la palabra, false si no existe

Definition at line 196 of file diccionario.h.

```
00196                                           {
00197             bool borrada = false;
00198             typename list<data<T,U> >::iterator it;
00199             if(Esta_Clave(clave,it)){
00200                 datos.erase(it);
00201                 borrada = true;
00202             }
00203             return borrada;
00204         }
```

Here is the call graph for this function:



3.2.3.13 removeSignificados_Palabra()

```
template<class T , class U >
bool Diccionario< T, U >::removeSignificados_Palabra (
            const T & clave )  [inline]
```

Elimina todos los significados de una palabra a partir de la clave Funcion extra "accidental" en el proceso de removePalabra.

Parameters

clave	La palabra a la que borrar todas sus definiciones, sin borrar la palabra

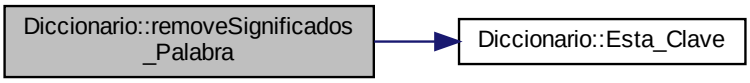
Returns

true si borra las definiciones, false en otro caso

Definition at line 179 of file diccionario.h.

```
00179                                                       {
00180             bool borrados = false;
00181             typename list<data<T,U> >::iterator it;
00182             cout « "Va a borrar las definiciones de " « clave « ":" « endl;
00183                 if (Esta_Clave(clave,it)){
00184                     (*it).info_asoci.clear();
00185                     borrados = true;
00186                 }
00187
00188             return borrados;
00189         }
```

Here is the call graph for this function:



3.2.3.14 size()

```
template<class T , class U >
int Diccionario< T, U >::size ( ) const  [inline]
```

Devuelve el tamaño del diccionario.

Returns

El tamaño del diccionario

Definition at line 297 of file diccionario.h.

```
00297                           {
00298                 return datos.size();
00299          }
```

The documentation for this class was generated from the following file:

- include/diccionario.h

Chapter 4

File Documentation

4.1 diccionario.h

```
00001 #ifndef _DICCIONARIO_H
00002 #define _DICCIONARIO_H
00003
00004 #include <iostream>
00005 #include <string>
00006 #include <list>
00007 using namespace std;
00008
00014 template <class T,class U>
00015 struct data{
00016     T clave;
00017     list<U> info_asoci;
00018  };
00019
00020
00025 template <class T, class U>
00026 bool operator< (const data<T,U> &d1,const data <T,U>&d2){
00027        if (d1.clave<d2.clave)
00028              return true;
00029        return false;
00030 }
00031
00039 template <class T,class U>
00040 class Diccionario{
00041       private:
00042
00043
00044             list<data<T,U> > datos;
00045
00046             void Copiar(const Diccionario<T,U>& D){
00047                /*typename list<data<T,U> >::const_iterator it_d;
00048                typename list<data<T,U> >::iterator it=this->datos.begin();*/
00049
00050                datos.assign(D.datos.begin(),D.datos.end());
00051                /*for (it_d=D.datos.begin(); it_d!=D.datos.end();++it_d,++it){
00052                      this->datos.insert(it,*it_d);
00053
00054                }*/
00055             }
00056
00057             void Borrar(){
00058
00059                   this->datos.erase(datos.begin(),datos.end());
00060             }
00061
00062
00063       public:
00067          Diccionario():datos(list<data<T,U> >()){}
00068
00073          Diccionario(const Diccionario &D){
00074                Copiar(D);
00075          }
00076
00080          ~Diccionario(){}
00081
00087          Diccionario<T,U> & operator=(const Diccionario<T,U> &D){
00088                if (this!=&D){
00089                      Borrar();
```

```
00090                   Copiar(D);
00091               }
00092               return *this;
00093           }
00094
00104           bool Esta_Clave(const T &p, typename  list<data<T,U> >::iterator &it_out){
00105
00106                if (datos.size()>0){
00107
00108                    typename list<data<T,U> >::iterator it;
00109
00110                    for (it=datos.begin(); it!=datos.end() ;++it){
00111                      if ((*it).clave==p) {
00112                            it_out=it;
00113                            return true;
00114                      }
00115                      else if ((*it).clave>p){
00116                            it_out=it;
00117                            return false;
00118                      }
00119
00120                    }
00121
00122                    it_out=it;
00123                    return false;
00124                }
00125                else {
00126                     it_out=datos.end();
00127                     return false;
00128                }
00129           }
00130
00138           void Insertar(const T& clave,const list<U> &info){
00139
00140               typename list<data<T,U> >::iterator it;
00141
00142               if (!Esta_Clave(clave,it)){
00143                     data<T,U> p;
00144                     p.clave = clave;
00145                     p.info_asoci=info;
00146
00147                     datos.insert(it,p);
00148
00149               }
00150
00151           }
00152
00160           void AddSignificado_Palabra(const U & s ,const T &clave){
00161                 typename list<data<T,U> >::iterator it;
00162
00163                 if (!Esta_Clave(clave,it)){
00164                      data<T,U> p;
00165                      p.clave = clave;
00166                      datos.insert(it,p);
00167                 }
00168
00169                  //Insertamos el siginificado al final
00170                     (*it).info_asoci.insert((*it).info_asoci.end(),s);
00171           }
00172
00179           bool removeSignificados_Palabra(const T & clave){
00180               bool borrados = false;
00181               typename list<data<T,U> >::iterator it;
00182               cout « "Va a borrar las definiciones de " « clave « ":" « endl;
00183                   if (Esta_Clave(clave,it)){
00184                       (*it).info_asoci.clear();
00185                       borrados = true;
00186                   }
00187
00188               return borrados;
00189           }
00190
00196           bool removePalabra(const T & clave){
00197               bool borrada = false;
00198               typename list<data<T,U> >::iterator it;
00199               if(Esta_Clave(clave,it)){
00200                   datos.erase(it);
00201                   borrada = true;
00202               }
00203               return borrada;
00204           }
00205
00211           Diccionario operator+(const Diccionario & d){
00212               Diccionario aux(*this);
00213               typename list<data<T,U»::iterator it2;
00214               typename list<data<T,U»::const_iterator it;;
00215               for (it=d.datos.begin();it!=d.datos.end();++it){
```

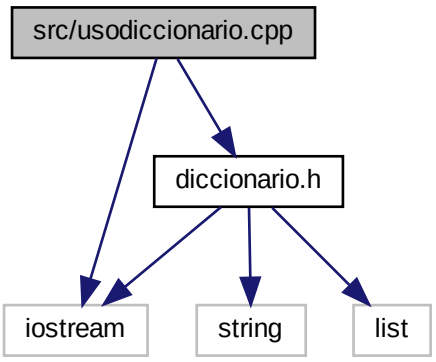
```
00216                 if(aux.Esta_Clave((*it).clave,it2)){ //mete las definiciones que no existan de la
      palabra
00217                     typename list<U>::const_iterator itinfod, itinfoaux;
00218                     bool existedef = false;
00219                     for (itinfod=(*it).info_asoci.begin();itinfod !=
      (*it).info_asoci.end();++itinfod){
00220                         existedef = false;
00221                         for(itinfoaux=(*it2).info_asoci.begin();itinfoaux != (*it2).info_asoci.end()
      && !existedef;++itinfoaux)
00222                             if((*itinfoaux) == (*itinfod))
00223                                 existedef = true;
00224                         if(!existedef)
00225                             aux.AddSignificado_Palabra((*itinfod),(*it).clave);
00226                     }
00227                 }
00228                 else //Si no existe la palabra, la mete
00229                     aux.Insertar((*it).clave,(*it).info_asoci);
00230             }
00231             return aux;
00232         }
00233
00240         Diccionario getElementosEntreClaves(const T & clavea, const T & claveb){
00241             Diccionario<T,U> D;
00242             typename list<data<T,U»::iterator ita, itb;
00243             bool terminar = false;
00244             int disa,disb;
00245             if (Esta_Clave(clavea, ita) && Esta_Clave(claveb, itb)){
00246                 disa = distance(datos.begin(), ita);
00247                 disb = distance(datos.begin(), itb);
00248                 if(disa <= disb){ //Si clave_a esta antes o es igual que clave_b
00249                     while (ita != itb){
00250                         D.Insertar((*ita).clave, (*ita).info_asoci);
00251                         ita++;
00252                         cout « (*ita).clave « " y " « (*itb).clave « endl;
00253                     }
00254                     D.Insertar((*ita).clave, (*ita).info_asoci); //Añadimos la clave b
00255                 }
00256                 else{ //Si clave_b esta antes que clave_a
00257                     while (itb != ita){
00258                         D.Insertar((*itb).clave, (*itb).info_asoci);
00259                         itb++;
00260                         cout « (*itb).clave « " y " « (*itb).clave « endl;
00261                     }
00262                     D.Insertar((*itb).clave, (*itb).info_asoci); //Añadimos la clave b
00263                 }
00264
00265             }
00266             else
00267                 cerr « "Una de las claves introducidas no existe" « endl;
00268
00269             return D;
00270         }
00271
00280           list<U> getInfo_Asoc(const T & p) {
00281                typename list<data<T,U> >::iterator it;
00282
00283                if (!Esta_Clave(p,it)){
00284                 return list<U>();
00285                }
00286                else{
00287                 return (*it).info_asoci;
00288                }
00289          }
00290
00291
00292
00297          int size()const{
00298                return datos.size();
00299          }
00300
00301
00310          typename list<data<T,U> >::iterator  begin(){
00311                return datos.begin();
00312          }
00317          typename list<data<T,U> >::iterator end(){
00318                return datos.end();
00319          }
00320
00325          typename list<data<T,U> >::const_iterator  begin()const{
00326                return datos.begin();
00327          }
00328
00333          typename list<data<T,U> >::const_iterator end()const {
00334                return datos.end();
00335          }
00336
00337 };
```

```
00338
00339 #endif
00340
00341
00342
00343
```

4.2 src/usodiccionario.cpp File Reference

```
#include <iostream>
#include "diccionario.h"
```

Include dependency graph for usodiccionario.cpp:



Functions

- ostream & operator<< (ostream &os, const Diccionario< string, string > &D)

  Operator<<. Obsérvese el uso de 2 tipos diferentes de iteradores. Uno sobre listas de listas y otro sobre listas.
- istream & operator>> (istream &is, Diccionario< string, string > &D)

  Operator >>. El formato de la entrada es: numero de claves en la primera linea clave-iésima retorno de carro numero de informaciones asociadas en la siguiente linea y en cada linea obviamente la informacion asociada.
- void EscribeSigni (const list< string > &l)

  Recorre la lista de información asociada a una clave y la imprime.
- void separador ()

  separa las salidas de texto para una mejor lectura
- int main ()

  Lee un diccioario e imprime datos asociados a una clave. Hay un fichero ejemplo de prueba: data.txt.Para lanzar el programa con ese fichero se escribe ./usodiccionario < data.txt.

4.2.1 Detailed Description

Author

Yeray Lopez Ramirez

Date

Diciembre de 2021

Definition in file usodiccionario.cpp.

4.2.2 Function Documentation

4.2.2.1 EscribeSigni()

```
void EscribeSigni (
            const list< string > & l )
```

Recorre la lista de información asociada a una clave y la imprime.

Parameters

l	La lista de definiciones

Definition at line 79 of file usodiccionario.cpp.

```
00079                                       {
00080       list<string>::const_iterator it_s;
00081       for (it_s=l.begin();it_s!=l.end();++it_s){
00082             cout<<*it_s<<endl;
00083       }
00084 }
```

4.2.2.2 main()

```
int main ( )
```

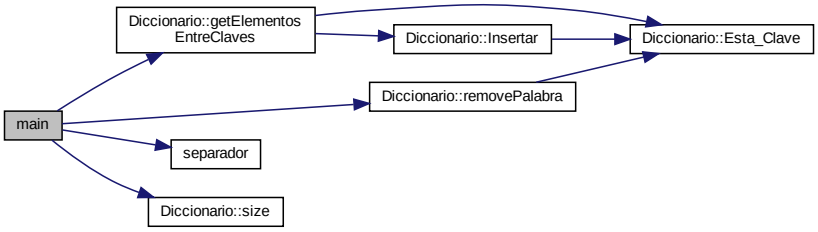
Lee un diccioario e imprime datos asociados a una clave. Hay un fichero ejemplo de prueba: data.txt.Para lanzar el programa con ese fichero se escribe ./usodiccionario < data.txt.

Definition at line 100 of file usodiccionario.cpp.

```
00100           {
00101     //Diccionario 1
00102     Diccionario<string,string> D1;
00103     cin>>D1;
00104     cout << "Diccionario 1:" << endl;
00105     cout<<D1;
00106
00107     if(D1.size() == 0)
00108         cout << "El diccionario 1 esta vacio" << endl;
00109     //Diccionario 2
00110     separador();
00111     Diccionario<string,string> D2;
00112     cin>>D2;
00113     cout << "Diccionario 2:" << endl;
00114     cout<<D2;
00115
00116     if(D2.size() == 0)
00117         cout << "El diccionario 2 esta vacio" << endl;
00118     //removePalabra: elimina una palabra del diccionario
00119     separador();
00120     cout << "Palabra a eliminar del Diccionario 1:" << endl;
00121     string a;
00122     cin>>a;
00123     D1.removePalabra(a);
00124     cout << "Diccionario despues de borrar " << a << endl;
00125     cout << D1;
00126
00127     //operator+: une dos diccionarios
00128     separador();
00129     cout << "La union de los dos diccionarios es " << endl;
00130     Diccionario<string,string> D3 = D1+D2;
00131     cout << D3;
00132
```

```
00133     //getDiccionarioEntreClaves: Devuelve elementos entre dos claves
00134     separador();
00135     cout « "Escribes las 2 claves entre las que buscar" « endl;
00136     string clave1, clave2;
00137     cin » clave1 » clave2;
00138     cout « "Los elementos entre las claves '" « clave1 « "' y '" « clave2 «
00139         "' en el diccionario fusionado es:" « endl;
00140     Diccionario<string,string> DEntreClaves = D3.getElementosEntreClaves(clave1,clave2);
00141     cout « DEntreClaves « endl;
00142
00143     /* //Para borrar definiciones. Funcion extra "accidental"
00144     string b;
00145
00146     cout«"Introduce una palabra"«endl;
00147     cin»b;
00148
00149     //Borra las definiciones de una palabra
00150     //D1.removeSignificados_Palabra(b);
00151
00152     /*
00153     list<string>l=D1.getInfo_Asoc(b);
00154
00155     //Imprime las definiciones de una palabra
00156     cout « "Las definiciones de " « b « " son:" « endl;
00157     if (l.size()>0)
00158         EscribeSigni(l);
00159     */
00160 }
```

Here is the call graph for this function:



4.2.2.3 operator<<()

```
ostream & operator<< (
            ostream & os,
            const Diccionario< string, string > & D )
```

Operator<<. Obsérvese el uso de 2 tipos diferentes de iteradores. Uno sobre listas de listas y otro sobre listas.

Parameters

os	El operador de salida
D	El diccionario a escribir

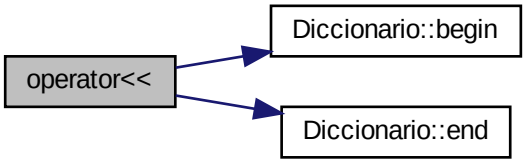
Returns

os La salida estandar

Definition at line 19 of file usodiccionario.cpp.

```
00019                                                                    {
00020
00021       list<data<string,string> >::const_iterator it;
00022
00023       for (it=D.begin(); it!=D.end(); ++it){
00024
00025             list<string>::const_iterator it_s;
00026
00027             os«endl«(*it).clave«endl«" informacion asociada:"«endl;
00028             for (it_s=(*it).info_asoci.begin();it_s!=(*it).info_asoci.end();++it_s){
00029                   os«(*it_s)«endl;
00030             }
00031             os«"************************************"«endl;
00032       }
00033
00034       return os;
00035 }
```

Here is the call graph for this function:



4.2.2.4 operator>>()

```
istream & operator>> (
            istream & is,
            Diccionario< string, string > & D )
```

Operator >>. El formato de la entrada es: numero de claves en la primera linea clave-iésima retorno de carro numero de informaciones asociadas en la siguiente linea y en cada linea obviamente la informacion asociada.

Parameters

is	El operador de entrada
D	El diccionario a leer

Returns

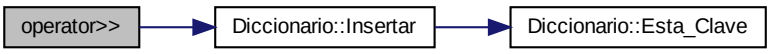
is La entrada estandar

Definition at line 47 of file usodiccionario.cpp.

```
00047                                                                    {
00048       int np;
00049       is»np;
00050       is.ignore();//quitamos \n
00051       Diccionario<string,string> Daux;
```

```
00052       for (int i=0;i<np; i++){
00053             string clave;
00054
00055             getline(is,clave);
00056
00057             int ns;
00058             is»ns;
00059             is.ignore();//quitamos \n
00060             list<string>laux;
00061             for (int j=0;j<ns; j++){
00062                   string s;
00063                   getline(is,s);
00064
00065                   // cout«"Significado leido "«s«endl;
00066                   laux.insert(laux.end(),s);
00067             }
00068             Daux.Insertar(clave,laux);
00069
00070       }
00071       D=Daux;
00072       return is;
00073 }
```

Here is the call graph for this function:



4.2.2.5 separador()

```
void separador ( )
```

separa las salidas de texto para una mejor lectura

Definition at line 89 of file usodiccionario.cpp.

```
00089                 {
00090     for(int i = 0; i < 100; i++)
00091         cout « "=";
00092     cout « endl;
00093 }
```

4.3 usodiccionario.cpp

Go to the documentation of this file.

```
00001
00007 #include <iostream>
00008 #include "diccionario.h"
00009
00010 //COMANDO EJECUCION: ./usodiccionario < ../datos/data.txt
00011
00019 ostream & operator«(ostream & os, const Diccionario<string,string> & D){
00020
00021       list<data<string,string> >::const_iterator it;
00022
00023       for (it=D.begin(); it!=D.end(); ++it){
00024
00025             list<string>::const_iterator it_s;
00026
00027             os«endl«(*it).clave«endl«" informacion asociada:"«endl;
```

```
00028             for (it_s=(*it).info_asoci.begin();it_s!=(*it).info_asoci.end();++it_s){
00029                   os«(*it_s)«endl;
00030             }
00031             os«"************************************"«endl;
00032       }
00033
00034       return os;
00035 }
00036
00047 istream & operator »(istream & is,Diccionario<string,string> &D){
00048       int np;
00049       is»np;
00050       is.ignore();//quitamos \n
00051       Diccionario<string,string> Daux;
00052       for (int i=0;i<np; i++){
00053             string clave;
00054
00055             getline(is,clave);
00056
00057             int ns;
00058             is»ns;
00059             is.ignore();//quitamos \n
00060             list<string>laux;
00061             for (int j=0;j<ns; j++){
00062                   string s;
00063                   getline(is,s);
00064
00065                   // cout«"Significado leido "«s«endl;
00066                   laux.insert(laux.end(),s);
00067             }
00068             Daux.Insertar(clave,laux);
00069
00070       }
00071       D=Daux;
00072       return is;
00073 }
00074
00079 void EscribeSigni(const list<string>&l){
00080       list<string>::const_iterator it_s;
00081       for (it_s=l.begin();it_s!=l.end();++it_s){
00082             cout«*it_s«endl;
00083       }
00084 }
00085
00089 void separador(){
00090     for(int i = 0; i < 100; i++)
00091         cout « "=";
00092     cout « endl;
00093 }
00094
00100 int main(){
00101     //Diccionario 1
00102     Diccionario<string,string> D1;
00103     cin»D1;
00104     cout « "Diccionario 1:" « endl;
00105     cout«D1;
00106
00107     if(D1.size() == 0)
00108         cout « "El diccionario 1 esta vacio" « endl;
00109     //Diccionario 2
00110     separador();
00111     Diccionario<string,string> D2;
00112     cin»D2;
00113     cout « "Diccionario 2:" « endl;
00114     cout«D2;
00115
00116     if(D2.size() == 0)
00117         cout « "El diccionario 2 esta vacio" « endl;
00118     //removePalabra: elimina una palabra del diccionario
00119     separador();
00120     cout « "Palabra a eliminar del Diccionario 1:" « endl;
00121     string a;
00122     cin»a;
00123     D1.removePalabra(a);
00124     cout « "Diccionario despues de borrar " « a « endl;
00125     cout « D1;
00126
00127     //operator+: une dos diccionarios
00128     separador();
00129     cout « "La union de los dos diccionarios es " « endl;
00130     Diccionario<string,string> D3 = D1+D2;
00131     cout « D3;
00132
00133     //getDiccionarioEntreClaves: Devuelve elementos entre dos claves
00134     separador();
00135     cout « "Escribes las 2 claves entre las que buscar" « endl;
00136     string clave1, clave2;
```

```
00137     cin » clave1 » clave2;
00138     cout « "Los elementos entre las claves '" « clave1 « "' y '" « clave2 «
00139         "' en el diccionario fusionado es:" « endl;
00140     Diccionario<string,string> DEntreClaves = D3.getElementosEntreClaves(clave1,clave2);
00141     cout « DEntreClaves « endl;
00142 
00143     /* //Para borrar definiciones. Funcion extra "accidental"
00144     string b;
00145 
00146     cout«"Introduce una palabra"«endl;
00147     cin»b;
00148 
00149     //Borra las definiciones de una palabra
00150     //D1.removeSignificados_Palabra(b);
00151 
00152     /*
00153     list<string>l=D1.getInfo_Asoc(b);
00154 
00155     //Imprime las definiciones de una palabra
00156     cout « "Las definiciones de " « b « " son:" « endl;
00157     if (l.size()>0)
00158         EscribeSigni(l);
00159     */
00160 }
00161 
```

Index