DICCIONARIOconPLANTILLAS

Practica3

Generado por Doxygen 1.7.5

Jueves, 30 de Octubre de 2014 09:22:02

Índice general

Capítulo 1

PRACTICA TEMPLATE

Versión

v0

Autor

Juan F. Huete

1.1. Introducción

En la práctica anterior se os pidió la implementación del tipo diccionario y un tipo meteorito. En esta práctica el objetivo es doble, por un lado permitir al alumno ver los errores cometidos en la práctica anterior y por otro lado seguir avanzando en el uso de las estructuras de datos, particularmente utilizando plantillas (templates) para la definición de tipos de datos genéricos, en este caso el diccionario.

La documentación se entrega mediante un fichero pdf, así como mediante un fichero zip que contiene todos los fuentes junto a los archivos necesarios para generar la documentación (en latex y html). Para generar los ficheros html con la documentación de la misma es suficiente con ejecutar desde la línea de comandos

```
doxygen dox_diccionario
```

Como resultado nos genera TRES directorios, uno con la documentación en html, otro con la documentación en latex y el último con la documentación en rtf que puede ser editado con office. Para generar la documentación en pdf podemos exportar el rtf a pdf desde el office o bien ejecutar los comandos de latex como sigue:

```
cd latex
make
```

Al hacer make se ejecuta una llamada al programa latex (si lo tenemos instalado) que como salida nos genera el fichero refman.pdf

Pasamos a detallar cada una de las partes de la práctica.

1.2. CORRECCION DE LA PRACTICA 2.

En decsai podeis encontrar los códigos fuentes de las dos versiones del tipo diccionario, junto con los fuentes de la clase meteorito, junto con el fichero principal.cpp que se ha diseñado para testear la validez de los métodos. Como antes, la compilación con una u otra versión se hace definiendo la constante DICC_V1 o DICC_V2 a la hora de compilar, esto es,

```
g++ -o ejecV1 -D DICC_V1 -std=c++0x fichero.cpp
g++ -o ejecV2 -D DICC_V2 -std=c++0x fichero.cpp
```

donde -o es el nombre del fichero ejecutable -D es la constante que nos indica la version con la que queremos compilar -std = c++0x nos indica que queremos utilizar c++ V11.

Los ficheros fuentes que se entregan

- diccionario.h Especificación del TDA diccionario.
- diccionarioV1.hxx primera versión del diccionario.
- diccionarioV2.hxx segunda versión del diccionario.
- meteorito.h especificacion de la clase meteorio
- meteorito.hxx implementación de la clase meteorito
- principal.cpp fichero de prueba del diccionario

1.2.1. SE PIDE

El alumno debe comparar su implementación con los distintos métodos entregados, hacer una crítica de los mismos, enviándome sus opiniones en un fichero pdf. En este caso, no es suficiente con indicar el hecho de que la implementaciones sean diferentes, sino en caso de serlo indicar cuáles han sido las dificultades encontradas a la hora de realizar la práctica.

Dicha crítica se debe entregar el Jueves 6 de Noviembre, a las 23:59 horas.

1.3. USO DE PLANTILLAS (TEMPLATES)

La segunda parte de la práctica está destinada al manejo de templates por parte del alumno. En concreto se pide dotar al diccionario la capacidad de poder definir un diccionario utilizando cualquier tipo de valores. Así, podriamos tener

```
#includde "diccionario.h"
...
// declaracion de tipos básicos:
diccionario<string,int> D1;
diccionario<int, string> D2;
diccionario<string,string> D3;
diccionario<float,string> D4;
```

```
// declaracion de tipos más complejos:

diccionario<nombreM,defM> meteoritos;
diccionario<string,list<nombreM> > tipos;

// declaracion de las entradas

diccionario<string,int>::entrada esimple1;
diccionario<nombreM,defM>::entrada emeteorito;
diccionario<string,list<nombreM> >::entrada etipos;


...

D1["Hola"]=3;
D3["Hola"]="tres";
D4[3.2]="tres con dos";

...
if (D1.find("Hola").second ==false) ....
```

En este caso, para realizar la práctica, el alumno deberá modificar tanto el fichero de especificación, diccionario.h, de forma que la propia especificación indique que trabaja con parámetros plantilla, como los ficheros de implementación (.hxx) de la clase diccionario.

De igual forma se debe modificar el fichero principal.cpp de manera que se demuestre el correcto comportamiento del diccionario cuando se instancia con distintos tipos. En concreto, se trabajara con otras dos instancias adicionales de diccionario:

- `diccionario<string,list<nombreM> > tipos;`

  que modela el diccionario que tiene como campo clave un string representando un código de meteorito y como definición tendremos la lista de todos los meteoritos que se clasifican bajo dicho tipo.

- `diccionario<string,int> met_anio;`

  que teniendo como clave un string que representa el año de caida, nos permitirá computar el número de meteoritos que han caido en dicho año.

Además del correcto funcionamiento de todos los métodos de la práctica anterior, se pide implementar los siguientes métodos en el fichero principal:

```
/** @brief organiza el conjunto de meteoritos por tipologia
@param[in] nombre_fichero que contiene el conjunto de meteoritos
@param[out] tipos un diccionario donde para cada tipo de meteorito se almacena la lista de todos los nom

Recordemos que el diccionario tiene clave unica, por tanto no puede haber elementos con  clave repetida.
Por este motivo, y dado que la tipologia del meteorito se repite por las distintas entradas del fichero,
se tendrá para cada una de las mismas una lista con los nombres de meteoritos.
*/

void listaTipos( const string  & nombre_fichero, diccionario<string,list<nombreM> > & tipos);



/** @brief Obtiene el diccionario teniendo en cuenta el año de caida del meteorito
```

```
@param[in] nombre_fichero que contiene el conjunto de meteoritos
@param[out] met_anio número de meteoritos clasificados por año de caida.
@return el año con más caidas.
*/

string caidos_por_anio(const string  & nombre_fichero, diccionario<string,int> & met_anio);
```

Como diccionario tiene sobrecargado el operator<<, para visualizar un diccionario cualquiera nos basta con sobrecargar el operator<< para cada una de los tipos claves y definición con que se instancie. En concreto, para nuestro caso, bastaría con sobrecargar el operator<< para el tipo list<nombreM> pues los otros ya están préviamente definidos.

1.3.1. SE PIDE

El alumno debe entregar los siguientes ficheros, con las correciones necesarias para poder trabajar con parámetros plantilla:

- diccionario.h Especificación del TDA diccionario genérica.
- diccionarioV1.hxx primera versión del diccionario genérico.
- diccionarioV2.hxx segunda versión del diccionario genérico.
- principalTemplate.cpp fichero de prueba del diccionario

Dicha entrega se debe realizar antes del Martes 11 de Noviembre, a las 23:59 horas.

Capítulo 2

Índice de clases

2.1. Lista de clases

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

Capítulo 3

Documentación de las clases

3.1. Referencia de la Clase defM

Métodos públicos

- **defM** (const defM &x)
- vector< string > **getCodes** () const
- bool **getFall** () const
- double **getLat** () const
- double **getLong** () const
- double **getMas** () const
- string **getYear** () const
- bool **setCode** (const string &s)
- void **setFall** (bool f)
- void **setLat** (double &lat)
- void **setLong** (double &longi)
- void **setMas** (const double &m)
- void **setYear** (const string &y)

Atributos privados

- vector< string > codes
- bool fall
- double lat
- double longi
- double mass
- string year

Amigas

- ostream & **operator**<< (ostream &, const defM &)

3.1.1. Documentación de los datos miembro

3.1.1.1. vector<string> defM::codes [private]

Vector de codigos

3.1.1.2. bool defM::fall [private]

true si Found, false si Fell

3.1.1.3. double defM::lat [private]

coordenada de latitud

3.1.1.4. double defM::longi [private]

coordenada de longitud

3.1.1.5. double defM::mass [private]

masa

3.1.1.6. string defM::year [private]

añi de caida, string

La documentación para esta clase fue generada a partir de los siguientes ficheros:

- meteorito.h
- meteorito.hxx

3.2. Referencia de la Clase diccionario

Clase diccionario.

`#include <diccionario.h>`

Tipos públicos

- typedef pair< nombreM, defM > **entrada**
- typedef unsigned int **size_type**

Métodos públicos

- diccionario ()

  constructor primitivo.
- diccionario (const diccionario &d)

  constructor de copia
- bool empty () const

  vacia Chequea si el diccionario esta vacio (size()==0)
- pair< diccionario::entrada, bool > find (const nombreM &s) const

  busca una meteorito en el diccionario
- bool insert (const entrada &e)

  Inserta una entrada en el diccionario.
- diccionario & operator= (const diccionario &org)

  operador de asignación
- defM & operator[] (const nombreM &s)

  Consulta/Inserta una entrada en el diccionario.
- const defM & operator[] (const nombreM &s) const

  Consulta una entrada en el diccionario.
- size_type size () const

  numero de entradas en el diccionario

Métodos privados

- bool cheq_rep () const

  Chequea el Invariante de la representacion.

Atributos privados

- vector< entrada > **dic**

Amigas

- ostream & operator<< (ostream &, const diccionario &)

  imprime todas las entradas del diccionario

3.2.1. Descripción detallada

Clase diccionario.

diccionario:: diccionario, find, operator[], size, Tipos diccionario::entrada, diccionario::size_type Descripción

Un diccionario es un contenedor que permite almacenar un conjunto de pares de elementos, el primero será la clave que deber ser única y el segundo la definición. En

nuestro caso el diccionario va a tener un subconjunto restringido de métodos (inserción de elementos, consulta de un elemento por clave, además de la consulta del elemento con mayor valor en la definición). Este diccionario "simulará" un diccionario de la stl, con algunas claras diferencias pue, entre otros, no estará dotado de la capacidad de iterar (recorrer) a través de sus elementos.

Asociado al diccionario, tendremos el tipo

```
diccionario::entrada
```

que permite hacer referencia al par de elementos almacenados en cada una de las posiciones del diccionario. Así, el primer campo de una entrada, first, representa la clave y el segundo campo, second, representa la definición. En nuestra aplicación concreta, la clave será un string representando una palabra válida del diccionario y el segundo campo es un entero que hace referencia a la frecuencia de ocurrencia de la palabra en el lenguaje.

El número de elementos en el diccionario puede variar dinámicamente; la gestión de la memoria es automática.

3.2.2. Documentación del constructor y destructor

3.2.2.1. diccionario::diccionario ( )

constructor primitivo.

Postcondición

define la entrada nula como el par ("",-1)

3.2.2.2. diccionario::diccionario (const diccionario & *d*)

constructor de copia

Parámetros

in	*d*	diccionario a copiar

3.2.3. Documentación de las funciones miembro

3.2.3.1. bool diccionario::cheq_rep ( ) const [private]

Chequea el Invariante de la representacion.

Devuelve

true si el invariante es correcto, falso en caso contrario

3.2.3.2. pair<diccionario::entrada,bool> diccionario::find (const nombreM & *s*) const

busca una meteorito en el diccionario

Parámetros

s	cadena a buscar

Devuelve

una copia de la entrada en el diccionario. Si no se encuentra devuelve la entrada con la definicion por defecto

Postcondición

no modifica el diccionario.

Uso

```
if (D.find("aaa").second ==true) cout << "Esta" ;
else cout << "No esta";
```

3.2.3.3. bool diccionario::insert (const entrada & *e*)

Inserta una entrada en el diccionario.

Parámetros

e	entrada a insertar

Devuelve

true si la entrada se ha podido insertar con éxito, esto es, no existe un meteorito con igual nombre en el diccionario. False en caso contrario

Postcondición

Si e no esta en el diccionario, el size() sera incrementado en 1.

3.2.3.4. diccionario& diccionario::operator= (const diccionario & *org*)

operador de asignación

Parámetros

in	*org*	diccionario a copiar. Crea un diccionario duplicado exacto de org.

3.2.3.5. defM& diccionario::operator[] (const nombreM & *s*)

Consulta/Inserta una entrada en el diccionario.

Busca la cadena s en el diccionario, si la encuentra devuelve una referencia a la definición de la misma en caso contrario la inserta, con una definición por defecto, devolviendo una referencia a este valor.

Parámetros

in	*s*	cadena a insertar
out	*defM*	& referencia a la definicion asociada a la entrada, nos permite modificar la definición

Postcondición

Si s no esta en el diccionario, el size() sera incrementado en 1.

3.2.3.6. const defM& diccionario::operator[] (const nombreM & *s*) const

Consulta una entrada en el diccionario.

Busca la cadena s en el diccionario, si la encuentra devuelve una referencia constante aa la definición de la misma, si no la encuentra da un mensaje de error.

Parámetros

in	*s*	cadena a insertar
out	*int*	& referencia constante a la definicion asociada a la entrada

Postcondición

No se modifica el diccionario.

3.2.3.7. size_type diccionario::size ( ) const

numero de entradas en el diccionario

Postcondición

No se modifica el diccionario.

3.2.4. Documentación de las funciones relacionadas y clases amigas

3.2.4.1. ostream& operator<< (ostream & , const diccionario &) [friend]

imprime todas las entradas del diccionario

Postcondición

No se modifica el diccionario.

Tareas pendientes implementar esta funcion

La documentación para esta clase fue generada a partir del siguiente fichero:

- diccionario.h