Esercizi di Programmazione ad Oggetti

Lista n. 3

Esercizio 1

Definire un template di classe contenitore `Dizionario<T>` i cui oggetti rappresentano una collezione di coppie (chiave,valore) dove la chiave è una stringa ed il valore è di tipo `T`. Ad una certa stringa può essere associato un solo valore di `T`. Si tratta quindi di definire un template di classe analogo al contenitore STL `map<string,T>`. Dovranno essere disponibili le seguenti funzionalità:

- inserimento: `bool insert(string, const T&)`
- rimozione: `bool erase(string)`
- ricerca per chiave: `T* findValue(string)`
- ricerca per valore: `vector<string> findKey(const T&)`
- overloading degli operatori di indicizzazione e output

Soluzione.

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

template <class T> class Dizionario;

template <class T>
ostream& operator<< (ostream&, const Dizionario<T>&);

template <class T>
class Dizionario {
  friend ostream& operator<< <T>(ostream &, const Dizionario<T>&);
public:
  bool insert(string, const T&);
  bool erase(string);
  T* operator[](string);
  bool operator==(const Dizionario<T>&) const;
  T* findValue(string);
  vector<string> findKey(const T&);
private:
  vector<string> chiavi;
  vector<T> valori;
  int index(string k); // -1 se la chiave k non e` presente
};

template <class T>
ostream& operator<< (ostream& os, const Dizionario<T>& diz){
  os << "diz:\n\t";
  for(int k=0; k<diz.chiavi.size(); ++k)
    os << diz.chiavi[k] << "-->" << diz.valori[k] << "\n\t";
  os << "------\n";
  return os;
}

template <class T>
int Dizionario<T>::index(string key){
  for(int k=0; k<chiavi.size(); k++)
    if(chiavi[k]==key) return k;
```

```
  return -1;
}

template <class T>
bool Dizionario<T>::insert(string key, const T& val)
{
  int idx=index(key);
  if(idx!=-1) return false;//chiave gia` presente
  else {
    chiavi.push_back(key);
    valori.push_back(val);
    return true;
  }
}

template <class T>
bool Dizionario<T>::erase(string key)
{
  int idx=index(key);
  if(idx==-1) return false;
  chiavi.erase(chiavi.begin()+idx);
  valori.erase(valori.begin()+idx);
  return true;
}

template <class T>
T* Dizionario<T>::operator[](string key){
  int idx=index(key);
  if(idx==-1) return 0;
  return &valori[idx];
}

template <class T>
vector<string> Dizionario<T>::findKey(const T& t){
  vector<string> v;
  for(int i=0; i<valori.size(); i++)
    if(valori[i]==t) v.push_back(chiavi[i]);
  return v;
}

template <class T>
T* Dizionario<T>::findValue(string key){
  for(int i=0; i<chiavi.size(); i++)
    if(chiavi[i]==key) // e' == su string
      return &valori[i];
  return 0;
}

template <class T>
bool Dizionario<T>::operator==(const Dizionario<T>& d) const {
    if(chiavi.size()!= d.chiavi.size())
        return false;
    for(int k=0; k<chiavi.size(); k++)
        if(!(d.findValue(chiavi[k])) || (*d[k] != chiavi[k]))
            return false;
    return true;
}
```

Esercizio 2

```
class Z {
public:
  ...
};

template <class T1, class T2=Z>
class C {
public:
  T1 x;
  T2* p;
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p);
  if(true == false) cout << ++(q->x);
  else cout << q->p;
  (q->x)++;
  if(*(q->p) == q->x) *(q->p) = q->x;
  T1* ptr = &(q->x);
  T2 t2 = q->x;
}

main(){
  C<Z> c1; fun(&c1); C<int> c2; fun(&c2);
}
```

Si considerino le precedenti definizioni. Fornire una dichiarazione (non è richiesta la definizione) dei membri pubblici della classe `Z` nel **minor numero possibile** in modo tale che la compilazione del precedente `main()` non produca errori. **Attenzione:** ogni dichiarazione in `Z` non necessaria per la corretta compilazione del `main()` sarà penalizzata.

```
class Z {
public:
  int operator++() {} // 1
  Z& operator++(int) {} // 2
  bool operator==(const Z& x) const {} // 3
  Z(int x=0) {} // 4
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p); // nulla
  if(true == false) cout << ++(q->x); // 1
  else cout << q->p;  // nulla
  (q->x)++; // 2
  if(*(q->p) == q->x) // 3
    *(q->p) = q->x; // nulla
  T1* ptr = &(q->x); // nulla
  T2 t2 = q->x; // 4 conversione T1 => T2
}
```

Soluzione.

```
class Z {
public:
```

```
  int operator++() {} // 1
  Z& operator++(int) {} // 2
  bool operator==(const Z& x) const {} // 3
  Z(int x=0) {} // 4
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p); // nulla
  if(true == false) cout << ++(q->x); // 1
  else cout << q->p;  // nulla
  (q->x)++; // 2
  if(*(q->p) == q->x) // 3
    *(q->p) = q->x; // nulla
  T1* ptr = &(q->x); // nulla
  T2 t2 = q->x; // 4 conversione T1 => T2
}
```