```
#ifndef ACCESS_H
#define ACCESS_H

#include "feat_skill.h"
#include "race.h"
#include "chars.h"
#include <set>
using namespace std;

/// Permessi di accesso, uso, ascolto...
class Access{
	private:
		/// Elenco feature obbligatorie
		set<Feat> yes_feat;
		/// Elenco feature vietate
		set<Feat> no_feat;
		/// Elenco skill con quantità minima
		set<Skill> yes_skill;
		/// Elenco skill con quantità massima
		set<Skill> no_skill;
		/// Elenco razze consentite
		set<Race> yes_race;
		/// Elenco razze vietate
		set<Race> no_race;
		/// Elenco appartenenze consentite
		set<Ownership> yes_own;
		/// Elenco appartenenze vietate
		set<Ownership> no_own;
	public:
		/// controlla permessi di personaggio giocante
		bool allowed(Character* pg);
};

#endif
```

```
#ifndef CHARS_H
#define CHARS_H

#include <string>
#include <set>
#include "race.h"
#include "tempset.h"
using namespace std;

/// Personaggio generico
class Character{
        private:
                /// Nome ed identificativo del personaggio
                string name;
                /// Breve descrizione
                string desc;
                /// Razza
                Race r;
                /// Classi di appartenenza (es. chierico, eterosessuale, juventino, ...)
                set<Ownership> owns;
                /// Lista Features
                TempSet<Feat> fe;
                /// Lista Skill
                TempSet<Skill> sk;
        public:
                /// Restituisce il nome
                string getName() {return name;}
                /// Restituisce la descrizione
                string getDesc() {return desc;}
                /// Restituisce la razza
                string getRace() {return r;}
                /// Restituisce il set delle classi di appartenenza
                set<Ownership>* getOwns() {return &owns;}
                /// Restituisce la lista delle features
                TempSet<Feat>* getFeats() {return &fe;}
                /// Restituisce la lista delle skill
                TempSet<Skill>* getSkills() {return &sk;}
};

/// Personaggio giocante
class PlayC : public Character{
};

#endif
```

```
#ifndef FEAT_SKILL_H
#define FEAT_SKILL_H

#include <string>
using namespace std;

/// Classe Skill per descrivere le abilità
/**
 * La classe Skill è composta di una stringa che descrive l'abilita e da un int
 * che la quantifica.
 * Ha un costruttore del tipo Skill("nuotare",10) ed una relazione d'ordine
 * che confronta il solo nome (identifica quindi abilità con diversi numeri)
 * per permetterne l'inserimento e la ricerca sulla base del nome in container
 * associativi.<br>
 * es. TempSet<Skill> t;<br>
 * t.equal_range(Skill("nuotare")) restituirà una coppia di iteratori che contengono
 * tutte le skill "nuotare" con relativi pesi.
 */
class Skill{
	private:
		string name;	///< Nome della skill
		int qu;		///< Peso della skill
	public:
		/// Restituisce il nome
		string n() const {return name;}
		/// Restituisce il peso
		int q() const {return qu;}
		/// Costruttore
		/**
		 * \param s il nome della skill (es. "nuotare")
		 * \param x il peso della skill (es. 10)
		 * il peso ha un default = 0 per poterlo omettere in fase di ricerca
		 */
		Skill(string s, int x=0) : name(s), qu(x) {}
		/// Ordinamento in base al solo nome
		friend bool operator<(const Skill& sx, const Skill& dx){
			sx.name<dx.name;
		}
};

// Le feature sono semplicemente stringhe
typedef string Feat;

#endif
```

```
#ifndef RACE_H
#define RACE_H

#include <string>
using namespace std;

// Race è semplicemente una stringa
typedef string Race;

// idem per Ownership
typedef string Ownership;

#endif
```

```
#ifndef TEMPSET_H
#define TEMPSET_H

#include <map>
#include <ctime>
#include <algorithm>
using namespace std;

template<class T> class TempSet;

/// Iteratore per TempSet
/**
 * L'iteratore è descritto da un puntatore al TempSet a cui si riferisce e da un iteratore
 * ad una multimap<T,time_t> che è contenuta nel TempSet.<br>
 * Sono realizzati i metodi minimi che ci servono: l'incremento, il confronto, la deferenziazione.
 */
template<class T> class TSiter{
        private:
                typedef multimap<T,time_t>::iterator ait;       ///< tipo dell'iteratore ausiliario
                typedef const T& reference_type;        ///< tipo reference
                TempSet<T>* cont;       ///< container corrente
                ait curr;       ///< itaratore ausiliario corrente

        public:
                /// costruttore banale
                TSiter() {};
                /// costruttore utile
                /**
                 * \param x il TempSet di riferimento
                 * \param v un iteratore alla multimap del TempSet di riferimento
                 */
                TSiter(TempSet<T>& x, ait v): cont(&x), curr(v) {};
                /// incremento (serve per scandire un intervallo)
                TSiter operator++();
                /// uguaglianza
                template<class C> friend bool operator==(const TSiter<C>& sx, const TSiter<C>& dx);
                /// deferenziazione
                reference_type operator*() const;
};

/// Multiset temporeneo
/**
 * La classe TempSet fornisce una versione temporanea del multiset.<br>
 * Insieme all'oggetto da inserire viene passata un'eventuale data di scadenza.<br>
 * Il TempSet si occuperà di ignorare gli oggetti scaduti e di rimuoverli
 * di tanto in tanto dalla memoria.<br>
 * La classe è formata da una multimap<T,time_t> usata per memorizzare le coppie (oggetto,scadenza).
 */
template<class T> class TempSet{
        private:
                /// l'incremento dell'iteratore deve essere friend
                friend TSiter<T> TSiter<T>::operator++();
                typedef TSiter<T> iterator;     ///<tipo iteratore
                typedef multimap<T,time_t>::iterator ait;       ///< tipo iteratore ausiliario
                /// multimap per memorizzare i dati
                multimap<T,time_t> TS;
                /// contatore di find e equal_range
                int count;
                /// numero di find da far passare prima di una pulizia
                static const int clean=50;
                /// toglie gli oggetti scaduti dalla memoria
                void purge();
        public:
                /// iteratore all'inizio del TempSet
                iterator begin();
                /// iteratore a dopo la fine del TempSet
                iterator end();
                /// cerca un oggetto
                /**
                 * \return l'iteratore all'oggetto trovato o end() se non trovato
                 */
                iterator find(const T&);
                /// cerca un intervallo di oggetti
                /**
                 * \return una coppia di iteratori al primo e subito dopo
                 * l'ultimo degli oggetti trovati. entrambi end() in caso di ricerca fallita
                 */
                pair<iterator, iterator> equal_range(const T&);
                /// inserisce un oggetto con eventuale scadenza
                void insert(const T&, time_t=0);
};
```

```
/// predicato controllo scadenza
/**
 * usato da TempSet<T>::purge() per controllare quali dati
 * rimuovere perché scaduti
 */
template<class C> class old : public unary_function< pair<const C,time_t>, bool>{
        public:
                /// controlla scadenza
                bool operator()(pair<const C,time_t> cop) {
                        return (cop.second!=0 && time(0)>cop.second);
                }
};

template<class T> TSiter<T>::reference_type TSiter<T>::operator*() const{
        // il valore è memorizzato come primo elemento della coppia corrente
        return curr->first;
}

template<class T> TSiter<T> TSiter<T>::operator++(){
        // incrementa punatore ausiliario
        ait it=++curr;
        // incrementalo ancora finché non punta ad un elemento non vecchio
        // o alla fine del container
        while(it!=cont->TS.end() && old<T>()(*it))
                ++it;
        curr=it;

}

template<class T> bool operator==(const TSiter<T>& sx, const TSiter<T>& dx){
        // due TSiter sono uguali se si riferiscono allo stesso
        // container con lo stesso iteratore ausiliario
        return sx.cont==dx.cont && sx.curr==dx.curr;
}

template<class T> TempSet<T>::iterator TempSet<T>::begin(){
        // prendi il primo elemento come iteratore ausiliario
        ait b=TS.begin();
        // incrementalo finché non punta ad un elemento non vecchio
        // o alla fine del container
        while (b!=TS.end() && old<T>()(*b))
                ++b;
        return TSiter<T>(*this, b);
}

template<class T> TempSet<T>::iterator TempSet<T>::end(){
        // restituisce l'iteratore alla fine del container
        return TSiter<T>(*this, TS.end());
}

template<class T> TempSet<T>::iterator TempSet<T>::find(const T& what){
        // se sono passati clean find() dall'ultima spurgata spurga
        if (++count%clean==0)
                purge();
        // cerca gli elementi cirrispondenti nella map
        pair<ait,ait> rang=TS.equal_range(what);
        // prendi il primo non vecchio
        ait ret=rang.first;
        while (ret!=rang.second &&  old<T>()(*ret))
                ++ret;
        // se esiste restituiscilo, altrimenti restituisci end()
        return ret==rang.second? TSiter<T>(*this, TS.end()): TSiter<T>(*this, ret);
}

template<class T> void TempSet<T>::purge(){
        // crea una copia di TS senza gli elementi vecchi
        multimap< T,time_t > purged;
        remove_copy_if(TS.begin(), TS.end(), inserter(purged,purged.end()), old<T>() );
        // sostituisci a TS la nuova map ripulita
        TS=purged;
}

template<class T> pair<TempSet<T>::iterator,TempSet<T>::iterator> TempSet<T>::equal_range(const T& w
hat){
        // se sporca da un po' pulisci la map
        if (++count%clean==0)
                purge();

        // cerca gli elementi corrispondenti
        pair<ait,ait> rang=TS.equal_range(what);

        // cerca il primo elemento non vecchio
```

```
        ait f=rang.first;
        while (f!=rang.second && old<T>()(*f))
                ++f;

        iterator fi,la;
        // se non ci sono elementi non vecchi
        if (f==rang.second){
                // restituisci end() come range
                fi=la=TSiter<T>(*this, TS.end());
        // altrimenti
        } else {
                // restituisci il primo buono come primo
                fi=TSiter<T>(*this, f);
                // e per l'ultimo controlla che non sia vecchio
                // e sostituiscilo col prox buono
                ait l=rang.second;
                while (l!=TS.end() && old<T>()(*l))
                        ++l;
                la=TSiter<T>(*this, l);
        }

        // restituisci il range calcolato
        return make_pair(fi,la);
}

template<class T> void TempSet<T>::insert(const T& what, time_t time=0){
        // inserisci una coppia
        // essendo TS una multimap non crea problemi di duplicazione di elementi
        // e possiamo lasciare insert come void
        TS.insert(make_pair(what,time));
}

#endif
```

```
#include "access.h"

bool Access::allowed(Character* pg){
        // leggi dati del personaggio
        TempSet<Feat>* c_feat=pg->getFeats();
        TempSet<Skill>* c_skill=pg->getSkills();
        set<Ownership>* c_owns=pg->getOwns();
        string c_name=pg->getName();
        string c_race=pg->getRace();

        // accesso
        bool ax=true;

        // controlla i permessi
        // features obbligatorie
        bool tmp=true;
        for(set<Feat>::iterator it=yes_feat.begin(); it!=yes_feat.end(); ++it)
                tmp=tmp && (c_feat->find(*it)!=c_feat->end());
        ax=ax && tmp;
        if (!ax) return false;

        // features vietate
        tmp=true;
        for(set<Feat>::iterator it=no_feat.begin(); it!=no_feat.end(); ++it)
                tmp=tmp && (c_feat->find(*it)==c_feat->end());
        ax=ax && tmp;
        if (!ax) return false;

        // razze consentite
        ax=ax && yes_race.find(c_race)!=yes_race.end();
        if (!ax) return false;

        // razze vietate
        ax=ax && no_race.find(c_race)==no_race.end();
        if (!ax) return false;

        // appartenenze consentite
        tmp=true;
        for(set<Ownership>::iterator it=yes_own.begin(); it!=yes_own.end(); ++it)
                tmp=tmp && (c_owns->find(*it)!=c_owns->end());
        ax=ax && tmp;
        if (!ax) return false;

        // appartenenze vietate
        tmp=true;
        for(set<Ownership>::iterator it=no_own.begin(); it!=no_own.end(); ++it)
                tmp=tmp && (c_owns->find(*it)==c_owns->end());
        ax=ax && tmp;
        if (!ax) return false;

        // skill con quantità minima
        tmp=true;
        for(set<Skill>::iterator it=yes_skill.begin(); it!=yes_skill.end(); ++it){
                // cerca la skill fra quelle del pg
                TempSet<Skill>::iterator sit=c_skill->find(*it);
                // se (non c’è oppure c’è ma ha quantità minore della minima)...
                if (sit==c_skill->end() || (*sit).q()<(*it).q())
                        tmp=false;
        }
        ax=ax && tmp;
        if (!ax) return false;

        // skill con quantità massima
        tmp=true;
        for(set<Skill>::iterator it=yes_skill.begin(); it!=yes_skill.end(); ++it){
                // cerca la skill fra quelle del pg
                TempSet<Skill>::iterator sit=c_skill->find(*it);
                // se (c’è e con quantità maggiore della massima)...
                if (sit!=c_skill->end() && (*sit).q()>(*it).q())
                        tmp=false;
        }
        ax=ax && tmp;

        return ax;
}
```

```
#include <string>
#include <iostream>
#include "tempset.h"
#include "feat_skill.h"
#include "race.h"
#include "chars.h"
#include "access.h"
using namespace std;

int main(){
	// esempio con features
	TempSet<Feat> tf;
	tf.insert(Feat("invulnerabilità"),time(0));
	tf.insert(Feat("ipi"),time(0)-1);
	tf.insert(Feat("ipi"),time(0)+2);
	tf.insert(Feat("ipi"),time(0)-1);
	tf.insert(Feat("blabla"),time(0));

	TempSet<Feat>::iterator it=tf.find(Feat("ipi"));
	if (it!=tf.end())
		cout << (*it) << endl;

	// esempio con skill
	TempSet<Skill> ts;
	ts.insert(Skill("scavare",2),time(0)+2);
	ts.insert(Skill("nuotare",3));    // senza scadenza
	ts.insert(Skill("nuotare",7),time(0)-2);          // scaduto
	ts.insert(Skill("nuotare",-5),time(0)+3);
	ts.insert(Skill("volare",7),time(0)+1);

	pair<TempSet<Skill>::iterator, TempSet<Skill>::iterator> cop2=ts.equal_range(Skill("nuotare"))
;
	for(TempSet<Skill>::iterator it=cop2.first; it!=cop2.second; ++it)
		cout << (*it).n() << " " << (*it).q() << endl;

	return 0;
}
```