Esercizio 1

Definire un template di classe `C<T,size>` con parametro di tipo `T` e parametro valore `size` di tipo intero che soddisfi le seguenti specifiche:

1. `MultiInfo<T>` è un template di classe associato ed annidato nel template `C<T,size>`. Un oggetto di `MultiInfo<T>` rappresenta un oggetto di tipo `T`, detto *informazione*, con una certa *molteplicità* `m` ≥ 0.

2. Un oggetto di `C<T,size>` rappresenta un array allocato dinamicamente di dimensione `size` di oggetti di `MultiInfo<T>`.

3. `C<T,size>` rende disponibile un costruttore `C(const T&, int)` con il seguente comportamento: una invocazione `C(t,k)` costruisce un oggetto di `C<T,size>` il cui array contiene in ogni posizione un oggetto di `MultiInfo<T>` con informazione `t` e quando `k` è ≥ 1 con molteplicità `k`, altrimenti (cioè quando `k` è < 1) con molteplicità 0.

4. Nel template `C<T,size>` il costruttore di copia, l'assegnazione e il distruttore devono essere "profondi", cioè la costruzione di copia e l'assegnazione di copia non devono provocare alcuna condivisione di memoria mentre la distruzione deve provocare anche la deallocazione di tutta la memoria dinamica.

5. `C<T,size>` rende disponibile l'overloading dell'operatore di indicizzazione `T* operator[](int)` con il seguente comportamento: se $0 \leq$ `k` $<$ `size` allora una invocazione `c[k]` ritorna un puntatore all'informazione di tipo `T` memorizzata nell'array di `c` in posizione `k`, altrimenti ritorna il puntatore nullo.

6. `C<T,size>` rende disponibile un metodo di istanza `int occorrenze(const T&)` con il seguente comportamento: una invocazione `c.occorrenze(t)` ritorna la somma delle molteplicità di tutte le occorrenze dell'informazione `t` nell'array memorizzato in `c`.

7. Deve essere disponibile l'overloading dell'operatore di output per oggetti di `C<T,size>` che permette di stampare tutte le informazioni di tipo `T` con relativa molteplicità memorizzate nell'array di un oggetto di `C<T,size>`.

Esercizio 2

```
class N;

class Smart {
public:
  N* p;
  Smart(N* q = 0): p(q) {if(p) p->counter++; cout << "Smart() ";}
  Smart(const Smart& x): p(x.p) {if(p) p->counter++; cout << "SmartCopy() ";}
  ~Smart() {
    if(p) {
      p->counter--; if(p->counter==0) delete p;
    };
    cout << "~Smart() ";
  }
  operator bool() {return p!=0;}
  Smart& operator=(const Smart& x) {
    Smart t = *this;
    p = x.p; if(p) p->counter++;
    cout << "Smart= ";
    return *this;
  }
  N* operator->() {return p;}
  bool operator==(const Smart& x) {return p==x.p;}
  bool operator==(N* x) {return p==x;}
};
class N {
public:
  string s;
  int counter;
  Smart next;
  N(string x, const Smart& y) : s(x), counter(0), next(y) {cout << "N() ";}
  ~N() {cout << "~N ";}
};
class C {
public:
  Smart punt;
  C(string s="BIANCO"): punt(new N(s,0)) {}
  void add(string s) {punt = new N(s,punt);}
  void modify() {
    if(punt && punt->next) punt->next = (punt->next)->next;
  }
  void print() {
    while(punt) { cout << punt->s << " "; punt=punt->next; }
  }
};
main() {
  C c1; cout << " **0" << endl;
  C c2("ROSSO"); cout << " **1" << endl;
  C c3(c2); cout << " **2" << endl;
  c3.add("VERDE"); cout << " **3" << endl;
  c3.add("BLU"); cout << " **4" << endl;
  c3.modify(); cout << " **5" << endl;
  c1=c3; cout << " **6" << endl;
  c1.print(); cout << " **7" << endl;
  c2.print(); cout << " **8" << endl;
  c3.print(); cout << " **9" << endl;
}
```

Il precedente programma compila (con gli opportuni `#include` e `using`) ed esegue correttamente. Si scrivano nelle apposite righe numerate le stampe prodotte dalla sua esecuzione e si usi l'ultima riga non numerata per le eventuali stampe successive alla stampa di `**9`. Se una riga non contiene alcuna stampa (oltre a quella già indicata) si scriva **NESSUNA STAMPA**.

..... `**0`

..... `**1`

..... `**2`

..... `**3`

..... `**4`

..... `**5`

..... `**6`

..... `**7`

..... `**8`

..... `**9`

.....

Esercizio 3

```
class N;

class S {
  friend ostream& operator<<(ostream&, const S&);      // (A)
  friend void stampa(N*);                              // (B)
private:
  string z;
public:
  S(string x = ""): z(x) {}
};
ostream& operator<<(ostream& os, const S& x) {return os << x.z;}

class N {
  friend class C;                                      // (C)
  friend void stampa(N*);                              // (D)
public:
  N* next;
private:
  S s;
  N(S t, N* p): s(t), next(p) {}

};
class C {
public:
  N* punt;
  C( N* x = new N(string("ROSSO"),0) ) : punt(x) {}
  void G() {if(punt) punt = punt->next;}
  void F(string s1, string s2 = "BLU") {
    punt = new N(s1,punt); punt = new N(s2,punt);
  }
};

void Fun(C* p1, C* p2) { if(p1 != p2) {*p1 = *p2; p1->G();} }
void stampa(N* p) { if(p) {cout << p->s << ' '; stampa(p->next);} }

main(){
  C* p = new C; p->F("VERDE");
  C* q = new C((p->punt)->next); q->F("BIANCO","NERO");
  stampa(p->punt); cout << "**1 " << endl;
  stampa(q->punt); cout << "**2 " << endl;
  C* t = new C(p->punt); Fun(p,q);
  stampa(p->punt); cout << "**3 " << endl;
  stampa(q->punt); cout << "**4 " << endl;
  Fun(q,t);
  stampa(p->punt); cout << "**5 " << endl;
  stampa(q->punt); cout << "**6 " << endl;
  q->F("GIALLO"); p->F("GIALLO");
  stampa(p->punt); cout << "**7 " << endl;
  stampa(q->punt); cout << "**8 " << endl;
}
```

A. Il precedente programma contiene le quattro dichiarazioni di amicizia `(A)`, `(B)`, `(C)`, `(D)`. Barrare con una croce nel riquadro sottostante le dichiarazioni di amicizia che sono **necessarie** per la corretta compilazione del programma (con gli opportuni `#include` e `using`):

(A)	**(B)**	**(C)**	**(D)**

B. Con le opportune dichiarazioni di amicizia, il precedente programma compila (con gli opportuni `#include` e `using`) ed esegue correttamente. Si scrivano nelle apposite righe numerate le stampe prodotte dalla sua esecuzione. Se una riga non contiene alcuna stampa (oltre a quella già indicata) si scriva **NESSUNA STAMPA**.

..... `**1`

..... `**2`

..... `**3`

..... `**4`

..... `**5`

..... `**6`

..... `**7`

..... `**8`