```
//
// APPELLO del 14/1/2022: Testo e possibili soluzioni
//

/* ESERCIZIO 1
 1.A (2 PUNTI) Realizzare due struct indirizzo e cliente:
 Indirizzo contiene: via, numero civico, CAP, città
 Cliente contiene: codice fiscale, cognome, nome, indirizzo
 */

struct indirizzo {
    string via;
    int numero_civico;
    string cap;
    string citta;
};


struct cliente{
    string cod_fiscale;
    string cognome;
    string nome;
    indirizzo ind; // era importante usare qui il tipo creato nella struct
     precedente
}

/*
 1.B (2 PUNTI) Realizzare una funzione che verifichi se due clienti abitano
  nella stessa zona della città (da verificare tramite il CAP)*/

// NB molti hanno letto il testo ma non hanno ragionato a fondo: la zona la
 verifichiamo col cap, ma la città deve essere comunque la stessa

// Attenzione agli argomenti della funzione: ci servono 2 cliente passati
 per valore o per rif costante

bool same_area(cliente c1, cliente c2) {
    return ((c1.citta == c2.citta) && (c1.cap == c2.cap));
}

/* ESERCIZIO 2

  Consideriamo il tipo di dato “coda di Elem”  e un’implementazione basata
  su vector

2.A (2 PUNTI) Produrre i prototipi (o interfacce) delle 3 funzioni
 principali
 – enqueue (inserisci elemento in fondo alla
 coda)
 – dequeue (elimina elemento dalla testa della
 coda)
 – front (accedi in lettura e restituisci il prossimo elemento nella coda)

*/
```

```
void enqueue(vector<Elem> &v, Elem e);
void dequeue(vector<Elem> &v);
Elem front(vector<Elem> v);

/*
 2.B (2 PUNTI) Implementare la funzione dequeue

 NB le code funzionano secondo il meccanismo FIFO (first in first out):
  esce dalla coda l'elemento che e' stato in coda piu' a lungo

 Una VERSIONE 1 qui di seguito permette di rispondere alla domanda in modo
  molto semplice (la difficoltà si sposterebbe sulla funzione enqueue, con
  inserimento di un elemento dalla testa)
 */

void dequeue(vector<Elem> &v){
    v.pop_back();
}

/* VERSIONE 2 - Alcuni hanno interpretato l'inserimento in testa, per
 esempio come nella versione qui di seguito.
 Hanno faticato in modo "inutile" ma va bene lo stesso! */

void dequeue(vector<Elem> &v){
    for (unsigned int i=0; i<v.size()-1; ++i)
        v.at(i)=v.at(i+1);
    v.pop_back();
}

/* ESERCIZIO 3

 Considerate le liste collegate semplici:
 typedef struct cell {
     int head;
     cell *next;
 } *lista;

 3.A (2.5 PUNTI) Realizzare una funzione ricorsiva che permetta di contare
  il numero di elementi di una lista
 */

int size(const lista l)
{
   if (l==nullptr)  // se la sequenza e' vuota la sua cardinalita' e' 0
      return 0;
   else
      return size(l->next)+1;
}


 /* 3.B (2.5 PUNTI) Realizzare una funzione booleana che restituisce true
  se tutti gli elementi della lista sono pari, false altrimenti.
    Trattare in modo opportuno il caso lista vuota
```

```
    (questo esercizio e' una "classica" visita di lista: analizzatelo in
     dettaglio)
  */

/* VERSIONE RICORSIVA */
bool all_even( lista l) {
    if (l == nullptr) return true; //tutti gli elementi di una lista vuota
     sono pari!
    if ((l->head%2)!=0) return false;
    return all_even(l->next);
}

/* VERSIONE ITERATIVA*/
bool all_even( lista l) {
    cell *cur=l;
    while (cur != nullptr) {
        if ((cur->head % 2)!=0) return false;
        cur=cur->next;
    }
    return true;
}

// Anche questa volta tanti errori sono stati causati da ECCEZIONI
 sollevate in modo non opportuno, per es nell'es 3
```