INFORMATICA – Preparazione alla prova di teoria

ESERCIZIO 1

Un negozio commercia prodotti ognuno dei quali ha un *nome*, costituito da un singolo carattere (maiuscolo o minuscolo), e un *prezzo unitario* in Euro. Ad esempio, il negozio potrebbe avere in catalogo il *prodotto di nome* `A` *con prezzo unitario* `15.5` *Euro* e il *prodotto di nome* `x` *con prezzo unitario* `30.15` *Euro*. La struttura

```
struct prodotto {
    char nome;
    float pu;
};
```

consente di definire variabili che memorizzano prodotti salvandone nome e prezzo unitario rispettivamente nei campi `nome` e `pu`. Supponendo che il negozio commerci esattamente `d` prodotti diversi, il catalogo del negozio può essere memorizzato nell'array `prodotto* C = new prodotto[d]`.

Supponete *già predisposto* il catalogo `C` (in altre parole, l'array `C` è già stato dichiarato e riempito con nome e prezzo unitario di tutti i `d` prodotti commerciati dal negozio) e assumete pure che NON esistano prodotti diversi con lo stesso nome.

Scrivete la funzione `float totaleCostoProd( prodotto* C, int d, char a, int b )` che restituisce il *costo totale* di `b` pezzi del prodotto di nome `a`, considerando il suddetto catalogo `C` di `d` prodotti. Se però il prodotto di nome `a` non esiste in catalogo oppure se il numero `b` di pezzi è un intero negativo, allora la funzione deve restituire `-1`.

SOLUZIONE

```
float totaleCostoProd( prodotto* C, int d, char a, int b ) {
    for ( int i = 0; i < d; i++ )
        if ( C[i].nome == a && b >= 0 )
            return C[i].pu * b;
    return -1;
}
```

ESERCIZIO 2

Un cliente del negozio all'ESERCIZIO 1 ordina online la spesa inserendo linea per linea il nome del prodotto da acquistare e, separato da un blank, il numero di pezzi di quel prodotto. Per convenzione, la spesa viene terminata inserendo un prodotto (sicuramente non in catalogo) dal nome `'#'` con una quantità qualsiasi. Ad esempio, il cliente potrebbe aver inserito le linee

```
A 2
B 9
x 9
# 7
```

perché intende comperare `2` pezzi del prodotto `A`, `9` pezzi del prodotto `B` e `9` pezzi del prodotto `x`; l'ultima linea contenete `#`, come detto sopra, segnala il termine della spesa.

Sfruttando la funzione `totaleCostoProd` dell'esercizio precedente, scrivete un frammento di codice che chieda al cliente la spesa linea per linea (se preferite, potete acquisire separatamente prima il nome del prodotto e poi il relativo numero di pezzi acquistato) e al termine stampi il *costo totale della spesa*. Ovviamente, nel conto finale, *non devono essere considerati prodotti non in catalogo o acquisti di prodotti in un numero negativo di pezzi.*

SOLUZIONE

```
char a;
int p;
float spesa = 0;
do {
    cout << "Inserire articolo e numero pezzi: ";
    cin >> a >> p;
    if ( totaleCostoProd( C, d, a, p ) != -1 )
        spesa = spesa + totaleCostoProd( C, d, a, p );
} while ( a != '#' );
cout << "Totale spesa: " << spesa << endl;
```

ESERCIZIO 3

Due array di interi si dicono *disgiunti* se non esiste alcun intero che compare in entrambi gli array. Scrivete la funzione `bool disjoint( int* X, int dx, int* Y, int dy )` che restituisce `true` qualora l'array `X` di dimensione `dx` sia disgiunto dall'array `Y` di dimensione `dy`, altrimenti restituisce `false`.

SOLUZIONE

```
bool disjoint( int* X, int dx, int* Y, int dy ) {
    for ( int i = 0; i < dx; i++ )
        for ( int j = 0; j < dy; j++ )
            if ( X[i] == Y[j] )
                return false;
    return true;
}
```

ESERCIZIO 4

Scrivete la funzione

`int most_frequent( int* A, int a )`

che restituisce uno degli interi *più frequenti* nell'array `A` di dimensione `a`.

SOLUZIONE

```
// numero di occorrenze di x in A
int occurrencies( int* A, int a, int x ) {
    int k = 0;
    for ( int i = 0; i < a; i++ )
        if ( A[i] == x )
            k++;
    return k;
}

// restituisce uno degli interi piu' frequenti in A
int most_frequent( int* A, int a ) {
    int mf = A[0];
    for ( int i = 1; i < a; i++ ) {
        if ( occurrencies( A, a, A[i] ) > mf )
            mf = A[i];
    return mf;
}
```