Esercitazioni di Tecniche di Programmazione

2. Seconda esercitazione autoguidata

due avvertenze:

1) Le soluzioni agli esercizi, le versioni di programmi dati nel testo delle esercitazioni e quant'altro sono raggiungibili tramite la pagina web del corso. Nel titolo di ogni sezione è specificato tra parentesi il nome del (o dei) file in cui è proposta una soluzione (se disponibile ...).
2) I programmi che scriveremo dovranno essere in accordo con la definizione standard **ANSI C** del linguaggio C; perciò, prima di cominciare vogliamo assicurarci che l'ambiente di programmazione che usiamo "usi" anche lui la medesima definizione.
    a. SE si usa il Dev C++, nella versione 4.9.9.2 (lingua inglese) bisogna andare nel menu' "Tools", selezionare "Compiler Options", scegliere "Settings" e poi "C Compiler" (selezionare almeno "Support all ANSI Standard C Programs")
    b. Se si usa il vecchio ed eroico Turbo C++, per essere sicuri di star usando ANSI C, bisogna assegnare opportunamente una certa opzione: aprire il menù OPTIONS, selezionare *Compiler* e poi *Source*. Nella finestra di scelta che appare, selezionate ANSI C.

2.1. Uso di `struct` (PUNTO.C, PUNTO2.C).

Scrivere un programma in cui

- sia definito il tipo `struct punto` adatto a rappresentare i punti colorati nel piano cartesiano (due dimensioni);
- vengano lette coordinate e colori di due punti e venga costruito il punto intermedio, stampandone i dati (il punto intermedia ha il medesimo colore dei due punti letti, se questi hanno il medesimo colore; altrimenti ha colore "NERO").

Scrivere poi il medesimo programma, in cui il tipo dei punti colorati sia TipoPunto, definito mediante typedef ed usato nelle dichiarazioni dei punti usati nel programma.

2.2. Uso di `struct` (DISTANZE.C).

Scrivere un programma in cui viene letta una sequenza di punti colorati (definiti come negli esercizi precedenti) e, per ogni punto letto, dal secondo in poi, viene stampata la *distanza colorata* tra esso e il precedente.

Ci sono dei colori prestabiliti come ammissibili: "`bianco`", "`rosso`", "`giallo`", "`celeste`", "`blu`", "`NERO`"
Quando viene letto il colore di un punto, se questo e' al di fuori dei colori prestabiliti, sara' considerato "`NERO`".

La *distanza colorata* tra due punti e' definita come la coppia
<`dl, dc`>, dove

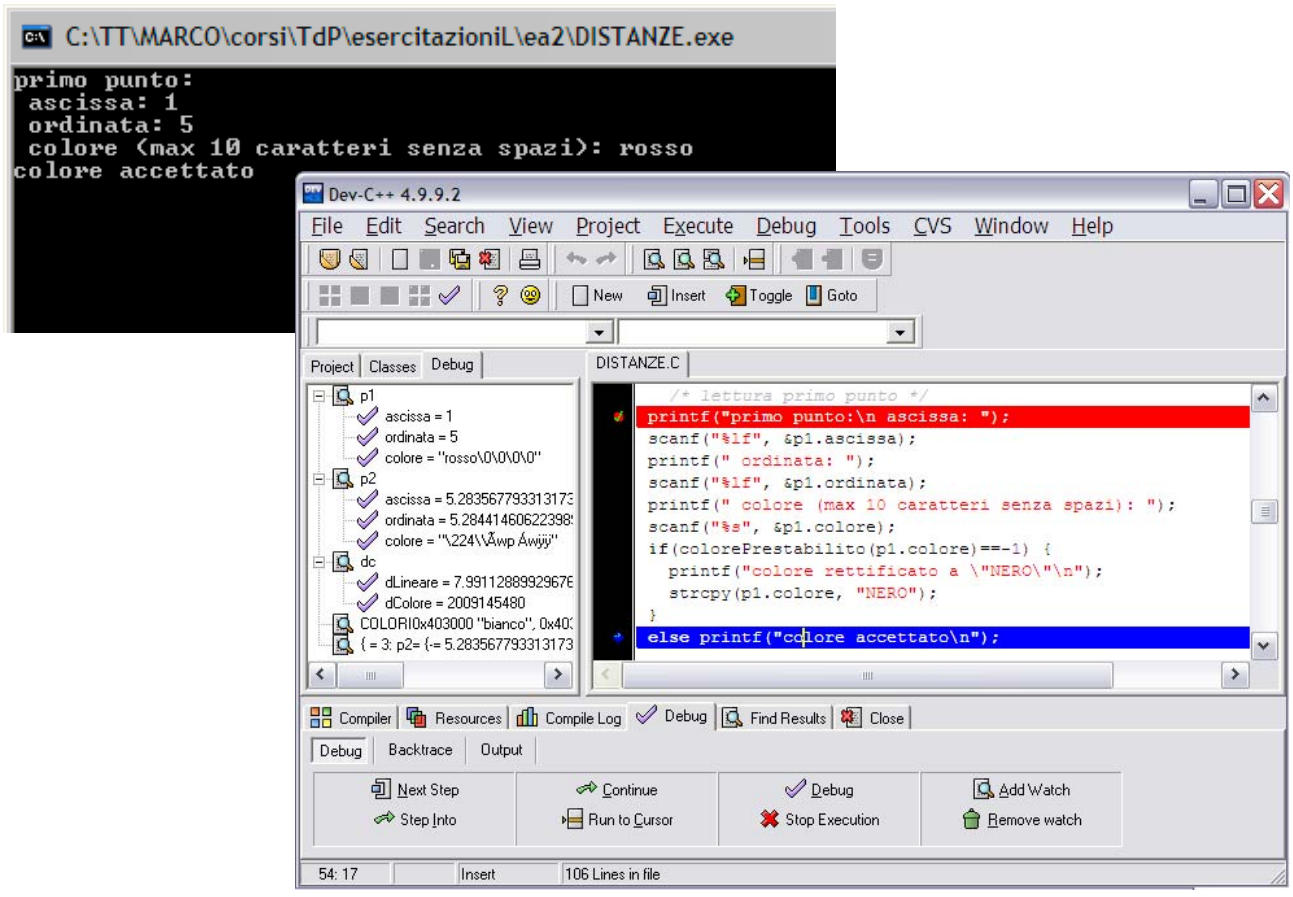
- dl=distanza lineare tra i due punti e
- `dc=distanza` tra i colori dei due punti (stabilita in base alla definizione dei colori prestabiliti data in precedenza: ad es. la distanza tra bianco e rosso e' 1, quella tra bianco e giallo e' 2, quella tra giallo e nero e' 4).

Qualche suggerimento per la soluzione segue: non li leggete tutti d'un fiato ...

Suggerimento 0

Quando il programma compila, probabilmente siamo solo a meta' dell'opera. Provare il programma anche usando le funzionalita' di debugging:
debugging?

- la linea rossa che si vede in figura e' una linea su cui e' stato impostato un breakpoint (basta ciccare accanto alla linea per impostare il breakpoint o per rimuoverlo).
- Un breakpoint nel punto indicato e' utile: cosi' possiamo scegliere l'opzione di compilazione con *debug* (F8) e usare F7 (*Next Step*) per far avanzare l'esecuzione del programma passo passo (un'istruzione alla volta).
- Durante l'esecuzione passo passo e' utile tener d'occhio il contenuto di alcune variabili significative: in figura si vede che e' stata impostata la visualizzazione del contenuto delle variabili p1, p2, dc: e' stato usato il comando *Add Watch*. (Nell'aggiungere una nuova watch, puo' essere necessario eseguire il passo successivo – F7– prima di vederla effettivamente.).
- La figura mostra un momento dell'esecuzione in cui tutte le istruzione di lettura del punto p1 sono state eseguite (la watch su p1 mostra quei valori: 1, 5, rosso).
- Se una watch corrisponde a variabili non nello scope dell'istruzione in esecuzione, cio' viene indicato (ad esempio nel punto del programma in esecuzione non potremmo vedere il contenuto della variabile i definita nella funzione colorePrestabilito().
- Quando e' in esecuzione un'istruzione di lettura, bisogna andare nella finestra di esecuzione e inserire il dato da leggere …
- Se non si vuole piu' procedere passo passo, Shift F7 (*Continue*) procura l'avanzamento dell'esecuzione fino alla prossima occorrenza di un breakpoint.



Suggerimento 1di5
la struttura del TipoPunto e' quella gia' vista; inoltre potrebbe convenire dare una definizione di un tipo apposito per la `DistanzaColorata`. (una struct con un campo double per la distanza lineare e uno int per la distanza tra i colori.

Inoltre e' bene che sia data una chiara definizione dei colori prestabiliti. Questa potrebbe anche non essere la definizione di un tipo vero e proprio. Una possibilita' e' quella di usare un array top level di stringhe inizializzato opportunamente `COLORI[]={bianco, rosso, giallo, celeste, blu, nero}`.

Suggerimento 2di5
All'array COLORI potrebbe essere affiancata una funzione `colorePrestabilito` che, ricevendo un colore, `col`, dica se `col` e' o no uno di colori ammessi. Nella soluzione proposta questa e' una funzione intera che restituisce –1 se col non e' ammissibile, oppure l'indice di `col` nell'array COLORI.

Suggerimento 3di5
La funzione `colorePrestabilito` (sempre nella soluzione proposta) viene usata anche per calcolare lo seconda parte della distanza colorata (infatti se il colore di `p1` e' `k1` e il colore di `p2` e' k2, la distanza tra i colori sara' `k1-k2` o `k2-k1` ...)

Suggerimento 4di5
Schema del programma

- lettura primo punto (p1)
- ciclo del tipo
```
while (continua!=0) {
      ...
}
```
con richiesta finale all'utente di inserire 1 se vuole continuare o 0 se vuole terminare (scelta che viene letta in `continua`).

Suggerimento 5di5

- durante ogni iterazione del ciclo viene
  - letto il punto p2
  - calcolata la distanza colorata tra p1 e p2 (e stampata)
  - assegnata continua opportunamente in base alla volonta' dell'utente
  - copiato p2 in p1, cosi' in p1 ci sara' il punto prima del prossimo letto

2.3. Funzioni che ricevono e/o restituiscono `struct` (PUNTO3.C).

Riscrivere il programma del punto 2.1, definendo ed usando le seguenti funzioni:

```
/*funzione che riceve un punto e lo stampa */
void stampaPunto(struct Punto p)

/* funzione che riceve due punti e ne restituisce
   il punto medio */
struct Punto puntoMedio(struct Punto pr, struct Punto sec)
```

2.4. Funzioni che ricevono puntatori a `struct` (PUNTO4.C).

Scrivere un programma che soddisfi i seguenti requisiti:

- il programma legge i dati relativi a due punti colorati (definiti come in precedenza)
- il programma calcola e stampa i dati relativi al punto medio tra i due letti da input;
- poi il programma chiede in input un colore e assegna tale colore a tutti e tre i punti, stampandone successivamente i dati in output.

Il programma deve far uso delle funzioni definite nell'esercizio precedente e della funzione
```
void cambiaColoreA(TipoPunto *p, char col[])
```
che, ricevendo un punto (o meglio, il suo indirizzo), e un colore (`col`), modifica il punto assegnandogli `col` come colore.

Suggerimento:

conviene prendere come punto di partenza il programma fatto per l'esercizio precedente, aggiungendo la nuova funzione e modificando opportunamente la `main()`.

2.5. Funzioni che ricevono puntatori a `struct` (PUNTO5.C).

Ripetere l'esercizio del 2.3, definendo e utilizzando, per la lettura di ciascun punto, una funzione `leggiPunto(...)`

Suggerimento 1di2: la funzione deve ricevere un punto e riempirlo con dati letti da input.

Suggerimento 2di2: quindi la funzione deve ricevre l'indirizzo del punto da riempire.

2.6. Funzioni che restituiscono puntatori a `struct` (PUNTO6.C).

Rifacendosi a quanto fatto in precedenza, scrivere una funzione creaPuntoMedio(...) che, ricevendo due punti, restituisce una struct appositamente allocata dinamicamente, contenente la rappresentazione del punto medio tra i due parametri.

Suggerimento 1di2: Si tratta di rifare l'esercizio precedente, ma sostituendo la funzione puntoMedio con quella qui richiesta (che restituisce un puntatore a struct Punto e non una struct Punto.

Suggerimento 2di2: ecco uno stralcio del programma PUNTO6.C in cui si vede come viene usata la funzione creaPuntoMedio:

```
  int main(){
   TipoPunto p1, p2,
        *pM;   /* puntatore per il punto medio */

   printf("primo punto:");
   leggiPunto(&p1);
   printf("secondo punto:");
   leggiPunto(&p2);

   pM = creaPMedio(p1, p2);
   /* la chiamata a pMedio restituisce il puntatore ad una struct
  che e' stata allocata appositamente, e che rappresenta il punto
  medio tra p1 e p2 */

 printf(" - adesso i tre punti sono:\n -  p1 = ");
```

2.7. Quadrilateri (QUADRI.C).

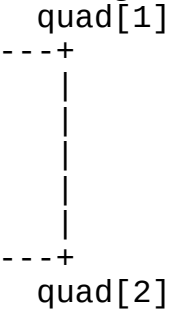
L'oggetto di questo esercizio e' un programma capace di gestire quadrilateri dati in input.
Un quadrilatero viene specificato come l'insieme dei suoi quattro vertici, che sono punti colorati sul piano (cioe' variabili del tipo `TipoPunto` definito precedentemente).

Supponiamo che i punti dati in input siano tutti distinti e diano luogo ad una figura in cui i lati sono "a 90 gradi".

Cio' assunto, il programma

- costruisce un quadrilatero allocando dinamicamente un array di quattro punti e leggendo i dati relativi ai quattro vertici;
- verifica se il quadrilatero e' un quadrato
- verifica se il quadrilatero e' isotetico, cioe' i suoi lati sono paralleli agli assi cartesiani;
- stampa i dati dei quattro vertici e stampa un messaggio in cui sia specificato se il quadrilatero e' o no un quadrato ed e' o no isotetico.

Se `quad` e' l'identificatore usato per il quadrilatero nel programma, si assume anche che i punti siano distribuiti come nel disegno qui sotto:



Usare nel programma le seguenti funzioni:

- `void stampaPunto(TipoPunto )` (per stampare i dati di un punto passato per parametro)
- `void leggiPunto( TipoPunto *)` (per leggere un punto)
- `void leggiQuadrilatero(TipoPunto q[4])` (per leggere l'intero quadrilatero, usando leggiPunto);
- `void stampaQuadrilatero(TipoPunto q[4])` (per stampare i dati del quadrilatero, ad esempio come sequenza dei punti che ne sono vertici);
- `double lunghezza(TipoPunto primo, TipoPunto secondo)` (per calcolare la distanza tra due punti)

Suggerimento 1di2:
per verificare che il quadrilatero `quad` sia un quadrato basta verificare che siano uguali le misure dei suoi lati (cioe' delle distanze tra vertici consecutivi: `quad[0]--quad[1], quad[1]--quad[2], quad[2]--quad[3], quad[3]--quad[0],`).
Per calcolare la lunghezza del lato `quad[k]quad[h]` si usa la funzione `lunghezza(...)`.

Suggerimento 2di2:
per verificare che i lati sono paralleli agli assi, bisogna verificare che i lati

- `quad[0]--quad[1]` e `quad[2]--quad[3]` sian paralleli all'asse delle ascisse
- `quad[1]--quad[2]` e `quad[3]--quad[0]` sian paralleli all'asse delle ordinate.